Marta Patrícia Teixeira Fernandes

Marta Patrícia Teixeira Fernandes Extensões para Design de Hardware Digital em Aplicações Aeroespaciais

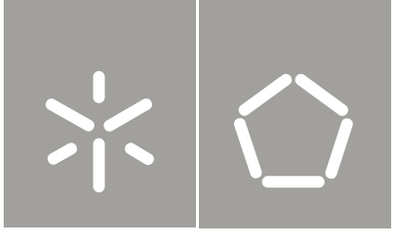
UMinho | 2012

Universidade do Minho
Escola de Engenharia

Marta Patrícia Teixeira Fernandes

Extensões para Design de Hardware Digital em Aplicações Aeroespaciais

Outubro de 2012



Universidade do Minho
Escola de Engenharia

Marta Patrícia Teixeira Fernandes

Extensões para Design de Hardware Digital em Aplicações Aeroespaciais

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia Comunicações

Trabalho efetuado sob a orientação do
Professor Doutor Paulo Cardoso

Outubro de 2012

Agradecimentos

Agradeço aos meus pais, Mário e Júlia Fernandes, e ao meu irmão Hugo, por todo o amor, carinho, apoio, dedicação e incentivo que me deram, por terem sempre acreditado em mim, e porque se não fosse por eles eu não estaria onde estou.
Ao meu orientador, professor Doutor Paulo Cardoso, ao professor Doutor Jorge Cabral, ao professor Doutor Adriano Tavares e ao professor Doutor João Monteiro, por toda a paciência, compreensão e excelente orientação, por me terem motivado em todos os momentos a realização e concretização desta dissertação e me permitiram caminhar na direção certa.
Ao Embedded System Research Group (ESRG) do departamento de Eletrónica Industrial da Universidade do Minho, que me acolheu e me proporcionou todas as condições necessárias para a elaboração deste trabalho. Agradeço em especial ao Paulo Garcia, Tiago Gomes, Filipe Salgado e Nuno Cardoso, por todo o carinho, amizade, paciência e pelos valiosos conhecimentos que me proporcionaram, sem eles nada disto seria possível.
Agradeço ao meu namorado, Artur Anjo, que sempre me apoiou e me ajudou nos momentos mais difíceis, pelo seu amor e amizade. Atodos os meus amigos, em especial, a Marta Carneiro, Carla Lima, Rita Silva, Sofia Correia, Ana Filipa Mangas, por estarem sempre presentes na minha vida. A minha amiga Natalia Antip, que é um exemplo para mim, com ela descobrir a minha verdadeira vocação.
A Lili ao Carlitos, a Ariana e a Bia pelo amor e amizade.
Aos meus colegas de curso e amigos, em especial para o Hugo Leite, que sempre me acompanhou no meu percurso académico e sem o apoio e ajuda dele tudo teria sido bem mais difícil.

Resumo

A comunicação é considerada por especialistas como uma necessidade básica para a sobrevivência dos seres humanos. Com os constantes avanços tecnológicos e conectividade eletrónica universal sente-se cada vez mais a necessidade e a importância do sigilo para a realização de operações de envio e receção de informação em aplicações, de forma a garantir segurança e fiabilidade da informação. Isto leva a uma maior consciência da necessidade de proteger dados e recursos de divulgação, para garantir a autenticidade da informação, e para proteger os sistemas de ataques na rede. Atualmente, dispositivos programáveis do tipo FPGA (*Field Programmable Gate Array*) são a principal opção para a implementação física de sistemas eletrónicos integrados, como tal diversas técnicas de tolerância a falhas têm sido propostas para a aplicação em FPGAs, por forma a tornar os sistemas confiáveis e com um alto desempenho, mesmo na ocorrência de falhas. O uso de HDLs (*Hardware Description Language*) para conceber implementações em FPGAs de elevada densidade é vantajoso, uma vez que as HDLs podem ser usadas para criar projetos grandes e complexos onde seja necessário que vários projetistas trabalhem em equipa, é possível que cada um possa trabalhar de forma independente em partes separadas de código.

A base deste trabalho consiste no estudo de diferentes HDLs, e identificar os pontos onde o nível de abstração pode ser aumentando. Identificando possíveis *constructs* que permitirão um desenvolvimento mais rápido e uma mais fácil compreensão por terceiros, surgindo uma futura extensão do SystemVerilog.

Nesta dissertação são implementadas novas metodologias para extensão da linguagem SystemVerilog tendo em conta o insuficiente nível de abstração na implementação de sistemas que requerem propriedades que implicam codificação minuciosa, como é o caso de sistemas com tolerância a falhas. É realizadoum *case study*utilizando HDLs,que consiste na implementação, em hardware, de um algoritmo de encriptação eficiente para

aplicações aeroespaciais com tolerância a falhas. A implementação, das técnicas de tolerância a falhas para aplicações aeroespaciais é essencial, devido a taxa de falhas por radiações cósmicas e ruido eletromagnéticoseja elevada no espaço quando comparada ao nível do mar, o que torna a análise das melhores técnicas de tolerância a falhas de suma importância.

Uma vez que não existe nenhuma técnica capaz de garantir que um sistema seja totalmente imune a falhas, torna-se necessário a análise de qual das técnicas aplicadas amenizará a vulnerabilidade com menores custos a nível de implementação e desempenho.

Abstract

Communication is considered by specialists as a basic need for the survival of human beings. With the constant advances in technology and universal electronic connectivity is perceived the growing need and importance of secrecy to conduct transmission and reception operations of information in applications, to ensure safety and reliability of information. This leads to a greater awareness of the need to protect data and resources from disclosure, to guarantee the authenticity of information, and to protect systems from network attacks. Nowadays, devices like programmable FPGA (Field Programmable Gate Array) are the main option for the physical implementation of integrated electronic systems, such as different fault tolerance techniques have been proposed for implementation on FPGAs, in order to make systems reliable and with high performance even in the occurrence of failures. The use of HDLs to design high density FPGAs implementations is advantageous since the HDLs can be used to create large and complex designs where it is necessary that several designers workas a team, it is possible that each can work independently on separate parts of code.

The basis of this work consists on the study of different HDLs, and identifies the points where the level of abstraction can be increased. Identifying possible constructs that enable faster development and more easily understood by others, creating a future extensionofSystemVerilog.

In this dissertation are implemented new methodologies for extension of SystemVerilog taking into consideration the insufficient level of abstraction in implementing systems that require properties which implymeticulous coding, as systems with fault tolerance. It is performed a case study using HDLs, which consists in the implementation in hardware of an efficient encryption algorithm for aerospace fault tolerant applications.

The implementation techniques of fault tolerance are critical for aerospace applications, because the failure rate for cosmic radiation and electromagnetic noise is high in space when compared to sea level, which makes analysis of the best techniques for fault tolerance of great importance.

Since there is not any technique which guarantees a system fully fault tolerant, it is necessary to analyze which the applied techniques will ease the vulnerability level with the lower costs implementation and performance.

Índice

Lista de Abreviaturas

ACE - Architecturally Correct Execution

AES - Advanced Encryption Standard

API – Application Program Interface

ASIC - Aplication-Specific Integrated Circuit

AVF - Architectural Vulnerability Factor

COTS – Comercial of-the-shelf

DES - Data Encryption Standard

FEC - Forward Error Correction

FPGA – Field Programmable Gate Array

HDL - Hardware Description Language

NSA - National Security Agency

IDEA - International Data Encryption Algorithm

NIST - National Institute of Standards and Technology

PLI – Program Language Interface

RC4 - Rivest Cipher 4

RS - Reed-Solomon

RLL – Rotate Left Logical

RRL – Rotate Right Logical

RTL – Register Transfer Level

SEC - Single Error Correction

SEC-DED - Single Error Correction Double Error Detection

SET's - Single-Event Transients

SEU's - Single-Event Upsets

SLA – ShiftLeft Arithmetic

SLL – Shift Left Logical

SRA – Shift Right Arithmetic

SRL – Shift Right Logical

SST/TLS - Secure Sockets Layer/Transport Layer Security

TLM - Transaction Level Modeling

TMR - Triple Modular Redundancy

UUT - Unit Under Test

VHDL - Very High Speed Integrated Circuit Hardware Description Language

WEP - Wired Equivalent Privacy

WPA - WiFi Protected Access

3DES - Triple Data Encryption Standard

Índice de Figuras

Índice de Tabelas

1. Introdução

1.1. Enquadramento

As atuais HDLs fornecem um insuficiente nível de abstração para permitir a eficiente implementação de sistemas complexos, principalmente quando o desenvolvimento é distribuído por diversos projetistas. Especialmente quando as implementações requerem propriedades que implicam codificação minuciosa, como por exemplo a implementação de sistemas com *Triple-Modular Redundancy* (TMR), os níveis de abstração oferecidos, restritos ao *Register Transfer Level* (RTL), limitam a produtividade do projetista bem como a compreensão do sistema por terceiros. Estas limitações tornam-se especialmente evidentes em aplicações onde a fiabilidade e segurança dos sistemas são extremamente importantes, como é o caso de aplicações aviónicas e aeroespaciais, onde se observa uma enorme necessidade de testes, verificação e reimplementação para se alcançar os objetivos do projeto.

1.2. Motivação

A comunicação sempre foi uma constante na vida humana, considerada por especialistas, como uma necessidade básica para sobrevivência e existência dos humanos. Vivemos numa Era onde os avanços tecnológicos trazem um enorme impacto na vida quotidiana, uma vez que existe uma conectividade eletrónica universal. Devido à existência de vírus, *hackers*, fraude e espionagem electrónica, sente-se cada vez mais a necessidade e a importância do sigilo para a realização de operações de envio e recepção de informação em aplicações, de forma a garantir segurança e fiabilidade da informação. Isto leva a uma maior consciência da necessidade de proteger dados e recursos de divulgação, para garantir a autenticidade da informação, e para proteger os sistemas de ataques.

Atualmente, dispositivos programáveis do tipo FPGA (*Field Programmable Gate Array*) são a uma importante opção para a implementação física de sistemas eletrónicosintegrados, onde a descrição do comportamento e da estrutura é feita usando linguagens de descrição de *hardware* (HDLs). O uso de HDLs para conceber implementações em FPGAs de elevada densidade é vantajoso, uma vez que as HDLs podem ser usadas para criar projetos grandes e complexos onde seja necessário que vários projetistas trabalhem em equipa, sendo possível que cada um possa trabalhar de forma independente em partes separadas de código. Além disso é possível verificar a

funcionalidade do projeto simulando o código RTL de forma a ser possível fazer alterações antes de este ser implementado ao nível da *gate*. O grande problema do uso de FPGAS é que viste que contém células SRAM e *flip-flops* isso os torna vulneráveis a falhas. As falhas mais comuns nestes tipos de aplicações são as falhas transitórias, que se podem classificar por SEUs (*Single-Event Upsets*), onde os bits nos flip-flops são alterados e por SETs (*Single-Event Transients*), onde os valores de lógica combinacional são temporariamente alterados. Devido a este tipo de falhas, diversas técnicas de tolerância a falhas têm sido propostas para a aplicação em FPGAs, por forma a tornar os sistemas confiáveis e com um alto desempenho, mesmo na ocorrência de falhas.As aplicações aeroespaciais e aviónicas são casos específicos de aplicações susceptíveis de sofrer falhas devido a causas naturais, principalmente a radiação cósmica e o ruído eletromagnético. Essas falhas podem ter consequências graves, tanto do ponto de vista económico como eventuais danos humanos, uma vez que se trata de aplicações onde a fiabilidade e segurança dos sistemas é extremamente importante.

É necessário utilizar mecanismos para lidar com os problemas que afetam o bom funcionamento das aplicações aeroespaciais, e consequentemente detetar e mitigar os efeitos dos SEUs e SETs, como tal é fundamental incorporar técnicas de tolerância a falhas as aplicações. O uso de tolerância a falhas torna necessário o uso de redundância, seja ela espacial, temporal ou de informação. No caso das aplicações aeroespaciais ou aviónicas, onde o desempenho, a área de silício e o consumo de energia assim como a fiabilidade e a segurança são cruciais para a viabilidade do seu desenvolvimento, a utilização de mecanismos de tolerância a falhas requer uma análise extensiva.

1.3. Objetivos

Nesta tese pretende-se estudar diversas HDLs (e.g., Verilog, VHDL, SystemVerilog,etc) e identificar os pontos onde se poderão definir *constructs* mais poderosos, i.e., que forneçam níveis de abstração mais elevados, consequentemente facilitando a compreensão. Estes deverão incidir sobre definição de interfaces entre módulos e encapsulamentos de módulos sequenciais/combinacionais e redundância a diversos níveis. Um *case-study* que necessite dos três pontos apresentados será desenvolvido e os pontos onde as necessidades apresentadas sejam identificadas serão estudados de modo a os necessários *constructs* serem modelados.

1.4. Estrutura da dissertação

Este trabalho está organizado em 5 capítulos.

No capítulo 1 é feita uma introdução do trabalho, onde é descritao enquadramento e a motivação para a elaboração deste trabalho. Também são referidos os objetivos principais e uma breve descrição da estrutura desta dissertação.

No capítulo 2 pretende-se apresentar o estado de arte, é neste capítulo que é feito um estudo das diversas HDLs e de extensões de linguagem de forma a ser possível identificar os pontos onde a abstração pode ser aumentada.

No capítulo 3, é feita uma breve descrição dos materiais e métodos usados na elaboração desta dissertação, além disso é apresentada a implementação da extensão á linguagem SystemVerilog elaborada consoante uma serie de especificações, assim como o caso de estudo elaborado que consistem em um sistema de encriptação para aplicações aeroespaciais com características de tolerância a falhas e com as novas metodologias implementadas.

No capítulo 4é feita uma discussão sobre os testes efetuados e os resultados obtidos.

Por ultimo, o capitulo 5 está reservado para as conclusões obtidas durante o desenvolvimento da dissertação

Existem ainda informações adicionais, relativas ao trabalhorealizado que se encontra na secção Anexos.

2. Análise e Estado da arte

Neste capítulo será feito um estudo de diversas linguagens de descrição de*hardware*, protocolos de encriptação e técnicas de tolerância a falhas, de forma a que após uma analise e estudo de aplicações já existentes no mercado seja possível decidir que protocolos ou técnicas usar para que se obtenha uma solução mais eficiente possível.

2.1. HDL (*Hardware Description Language*)

Uma linguagem de descrição de *hardware* (HDL) [16] é uma linguagem utilizada para descrever formalmente circuitos eletrónicos. As HDLs além de efetuarem descrição, também são utilizadas para projetar, testar e avaliar os circuitos. Possibilitam simulação de componentes de *hardware* com características reais, a nível de software, antes de serem construídos na prática, para que deste modo seja possível detetar e corrigir possíveis erros e assim reduzir os custos do projeto.A sintaxe e a semântica das HDLs incluem uma noção implícita de tempo, que é um atributo primário de *hardware*.

As HDLs são muito semelhantes às linguagens de programação, mas o seu uso é específico à descrição de estruturas e do comportamento de *hardware*. As linguagens HDL têm uma grande vantagem em relação ao desenvolvimento diretamente em portas lógicas, uma vez que elas podem representar diretamente equações booleanas, tabelas de verdade e operações complexas de um modo muito mais simplificado.

Uma linguagem HDL também pode ser utilizada na descrição em vários níveis de um circuito em desenvolvimento. As HDLs podem refinar e particionar uma descrição de alto nível, para converte-la em outras de nível mais baixo durante o processo de desenvolvimento. A descrição final deve conter todos os componentes primitivos e blocos funcionais.

De seguida são apresentadas várias linguagens de descrição de *hardware* muito utilizadas atualmente. Após um breve histórico, serão apresentadas as principais características de cada uma delas.

2.1.1. VHDL (*Very High Speed Integrated Circuit Hardware Description Language*)

A linguagem VHDL foi desenvolvida como um padrão que descreve a estrutura e funcionamento de circuitos integrados digitais. Como se trata de um padrão desenvolvido pelo IEEE, a cada 5 anos é feita a sua revisão podendo haver modificações no padrão inicialmente proposto. A utilização de VHDL permite que sistemas complexos possam seguir o desenvolvimento das etapas desse sistema, além disso permitea descrição do sistema em partes, e a decomposição de um sistema em subsistemas indicando como é que os subsistemas estão conectados.

Em VHDL existem duas abordagens para a descrição, a descrição comportamental e a estrutural.

A descrição comportamental de um sistema, pode ser vista como um caixanegra, que não se sabe o que faz, apenasque implementa uma determinada função, onde essa função será um conjunto de instruções escritas em VHDL, de nível mais ou menos elevado, que em termos de programação é válido, mas que num circuito pode ser impossível de sintetizar.

Na descrição estrutural, o sistema pode ser visto como uma interligação de componentes, que podem ser blocos construídos a partir de circuitos mais básicos.

De uma forma mais simples a estrutura de um programa em VHDL é composta por três elementos, como podemos ver na figura 2.1.

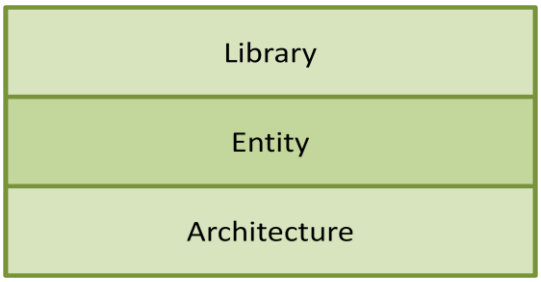


Figura 2.1– Elementos da estrutura de um programa VHDL

Library

Nos programas VHDL é necessário a declaração de *libraries* (bibliotecas) que serão usadas no projeto, uma vez que contém as primeiras informações contidas no programa.

Nas bibliotecas são armazenadas diversas funções e tipos básicos.

Usam-se *packages* quando se precisa utilizar um comando que não existe nas bibliotecas padrão, e devem ser definidos antes do início de uma *entity*.

Entity

A *entity*(entidade) é a parte principal do projeto, uma vez que define a interface (*port*) do projeto que descreve as entradas (*in*), as saídas (*out*) e o tipo do sinal correspondente.

A *entity* é composta por parâmetros e conexões, e tudo que é descrito na *entity* fica automaticamente visível a outras unidades associadas a esta.

entity*nome_da_entity***is**
port (
Declaração dos pinos
);
end [*nome_da_entity***] ;**

Arquitecture

A *arquitecture* é o corpo do sistema, define a lógica do sistema, e é aqui que são feitas as atribuições, comparações, operações, etc.

Uma *entity* pode ser formada por mais do que um *architecture*.

Architecture *nome_da_architecture* **of** *nome_da_entity* **is**
Declarações opcionais **(component e signal)**
begin
end [*nome_da_architecture***];**

A *arquitecture* pode ser composta pelos elementos que aparecem na figura 2.2.

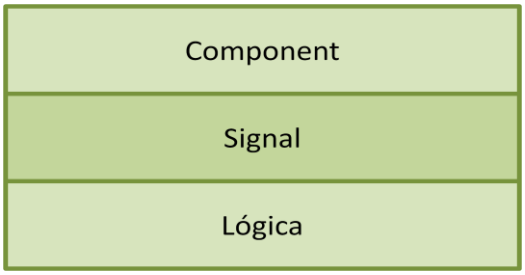


Figura 2.2– Elementos que compõem uma *arquitecture*.

A declaração de uma componente deve ser projetada através de outro programa VHDL. OS sinais servem para a comunicação entre os módulos e podem ser declarados em *entity*, *arquitecture*ou *package* (falaremos dele mais adiante). A lógica usa o comando *process*, como o seguinte formato:

Process (*lista de sensibilidade* **) begin**
descrição lógica
end process;

A lista de sensibilidade é composta por todos os sinais de entrada para os circuitos combinatórios e corresponde aos sinais que devem alterar a saída do circuito,

A seguir vão ser referidos os principais aspetos a ter em conta da linguagem VHDL.

Tipos

A linguagem VHDL possui um conjunto de tipos de estruturas de dados definidos, que são caracterizados por um conjunto de valores que podem assumir assim como o conjunto de operações que podem ser executados sobre eles.

Devem ser declarados como um tipo definido os objetos "*sinal*", "*variable*" e "*constant*" para que seja possível identificar os seus possíveis valores e as operações que podem ser realizadas sobre eles.

Os tipos de estrutura de dados escalares são tipos que não têm elemento ou estrutura interna, e todos os seus valores são ordenados e encontram-se numa faixa específica, ou então são explicitamente listados. Alguns tipos escalares já definidos na linguagem VHDL são apresentados na tabela 2.1

Tabela 2.1 - Tipos escalares da linguagem VHDL.

Tipo	Exemplo	Observação
BIT	0 ou 1	Valores lógicos 0 ou 1
BOOLEAN	True ou false	Usado em testes de decisão
INTEGER	De -2147483647 a 2147483647	Números inteiros
REAL	De -1.0E308 a 1.0E308	Números em ponto flutuante
CHARACTER	"0", "+", "A", "\"	Números, letras e caracteres

É possível que o utilizador defina algum outro tipo escalar, o que se dá o nome de *user defined type*. E no caso de ser necessário especificar valores físicos, usa-se o tipo *physical*, que se encontra previamente definido em VHDL mas somente para tempo.

Arrays

Arrays são estruturas regulares consistindo de elementos do mesmo tipo. Em VHDL existem dois tipos de *arrays*: *bit-vector*, que são elementos do tipo bit e *string*, que são elementos do tipo *character*.

Operadores

Existe uma serie de operadores na linguagem VHDL que é importante referir.

Operadores Lógicos – A maioria dos operadores usados são lógicos, uma vez que os sinais em VHDL são tipicamente lógicos. Temos os operadores *and*, *or*, *nand*, *xor* e *xnor* que exigem dois operandos, enquanto que o *not* apenas exige um.

Deslocamento –As operações de deslocamento são restritas a arrays, onde os elementos devem ser bit ou boolean. As operações de deslocamento são as seguintes, deslocamento lógico à direita (srl) e à esquerda (sll), deslocamento aritmético à direita (rla) e à esquerda (sla) e rotação lógica à direita (ror) e à esquerda (rol).

Operadores Relacionais –Este tipo de operadores são de atribuição e comparações entre dois objetos, que pode ser de igualdade (=), diferença (/n), menor (<), maior (>), menor ou igual (<=) e maior ou igual (>=).

Operadores Numéricos–Operadores de adição (+), subtracção (-), multiplicação (*), divisão (/), módulo (mod), valor absoluto (abs), resto (rem) e potência (**) tem que ser do mesmo tipo, e estão restritos aos tipos *integer* e real.

Decisões

Existem cinco comandos condicionais na linguagem VHDL, que são: *if then, if then else, case, for, loop* e *next.*

A condição ***if then*** executa o que estiver dentro do bloco se a condição for verdadeira. Na condição ***if then else*** executa o que estiver dentro do bloco *then* se a condição for verdadeira, senão é executado o bloco *else*. O ***case*** é usado quando o teste de condição de uma variável pode assumir várias opções. No caso do ***loop*** é executado quando um contador estiver dentro de uma faixa especificada. E por ultimo, o comando ***next*** é usado quando se deseja saltar determinados comandos para ir para um especifico.

TestBench

Na linguagem VHDL é possível testar um projeto, usando o TestBench onde os projetos são denominados de *Unit Under Test*(UUT). O projeto para ser testado usa sinais ou estímulos, para tal monitoria as respostas que obtêm e com essas respostas é feita uma melhor análise do *design*. O TestBench consiste em um *socket* para o UUT, em um gerador de sinais e ferramentas para monitorizar as respostas.

As formas de onda nos tesbench são usadas para simular cada módulo que é desenhado, é rápido e fácil de usar, visto que basta escolher as entradas que se desejam ver, e correr a simulação, desta forma os valores de saída vão também aparecer como forma de onda.

De uma forma geral, a linguagem VHDL traz a vantagem de permitir a simulação do sistema antes da sua implementação, de modo a que a simulação é compatível com o comportamento real. Além disso traz vantagens porque o projecto elaborado é independente da tecnologia que se queira utilizar, é possível a modularização e reutilização de *software*, entre outros.
A grande desvantagem da linguagem VHDL é que gera *hardware* não otimizado, isso faz com que a geração seja realizada de forma automática pelas ferramentas de síntese, o que provoca com que o processo de verificação e validação do hardware gerado se torne muito trabalhoso.

2.1.2 Verilog

A linguagem Verilog [17] [18] é uma linguagem de descrição de *hardware*, introduzida pela Gateway Design Automation, e que mais tarde foi comprada pela empresa Cadence Design Systems que tornou a linguagem Verilog de domínio público. Hoje em dia, a linguagem Verilog é um padrão IEEE.
A linguagem Verilog oferece ao *designer* todos os meios para descrever um sistema digital em vários níveis de abstração, além disso suporta ferramentas de projeto para síntese lógica. Os *designers* de *hardware* podem demonstrar as suas ideias com construções comportamentais, deixando os detalhes para fases posteriores do projeto. Uma representação abstrata pode ser utilizada para explorar alternativas arquiteturais através de simulações e para detetar restrições de projeto antes do projeto detalhado.
De forma a detetar possíveis restrições do um projeto, antes de este ser detalhado, é possível explorar alternativas arquiteturais através de simulação, usando uma representação abstrata. Após serem obtidos os detalhes do projeto, é possível criar descrições com construções estruturais.
A linguagem Verilog é bastante semelhante a linguagem de programação C, mas a sua construção básica é o módulo, e geralmente existe apenas um módulo por ficheiro, onde o *top level* do ficheiro invoca instâncias de outros módulos. Os módulos podem ser de especificação comportamental ou estrutural. Os módulos de especificação comportamental definem o comportamento do sistema digital, enquanto que os módulos de comportamento estrutural definem interconexões hierárquicas dos sub-módulos. Os sinais de entrada (input) e saída

(output) de um módulo são designados usando portos. O wire é um tipo de dados muito importante, que liga dois pontos, este tipo de dados pode ser usado para ligar por exemplo uma porta de saída a um driver, isto significa que o wire é usado para cria lógica combinacional, visto que este tipo de lógica não pode guardar um valor.

Os procedimentosem Verilog são comandos*always* ou *initial*, tarefas ou funções. Dentro de um bloco sequencial, todos os comandos devem aparecer entre um *begin* e um *end*, fazem parte de um procedimento, e são executados sequencialmente, mas os procedimentos em que são executados concorrentemente com outros procedimentos.

Para que um bloco seja executado repetidas vezes, é necessário que um bloco sequencial apareça em um comando *always*. Para que um bloco seja executado apenas uma vez, no início de uma simulação, o bloco sequencial é especificado com um comando *initial*.

Uma tarefa (task) é um procedimento chamado por outro, ou seja uma tarefa pode chamar outras tarefas e/ou funções, e como tal têm entradas e saídas, mas não tem um valor de retorno.

Por sua vez, uma função (*function*), é um procedimento usado em qualquer expressão, como tal não pode chamar uma tarefa, não tem saída mas retorna um valor único.

A linguagem Verilog permite que uma atribuição seja atrasada até que um evento específico ocorra, ou seja possui controlo de temporização, e pode ser especificado por um @.

A linguagem Verilog, ao contrário de VHDL, faz distinção entre o uso de maiúsculas e minúsculas. Além disso os identificadores podem conter qualquer sequencia de letras, dígitos, do símbolos $ e _, onde o primeiro carácter deve ser uma letra ou o símbolo _.

A seguir serão explicadas várias funcionalidades a ter em conta presentes na linguagem Verilog.

Instanciaçãoda entidade

Na linguagem VHDL o desenho da entidade pode ser instanciado na linguagem Verilog como se fosse um módulo equivalente do mesmo nome. Se a entidade tiver diversas opções de arquiteturas, a arquitetura utilizada é a padrão a não ser que se especifique outra.

VHDL:

```
entity myentity is
        port ( a, b : in std_logic;
        c : out std_logic);
end myentity;
Architecture implementacao of myentity
        ….
endimplementacao;
```

Verilog:

```
module myentity (a, b, c);
        input a, b;
        output c;
        wire a, b, c;
        ….
endmodule
```

Conjunto de valores

O padrão IEEE define uma lógica de quatro valores com quatro estados, 0, 1, Z e X.

O valor 0 representa o zero lógico, ou uma condição falsa, enquanto que o valor 1 representa o um lógico, ou uma condição verdadeira, ambos completam-se um ao outro. O valor z representa um estado de alta impedância, este valor encontra se presente na entrada de uma porta, ou quando ele é encontrado em uma expressão. O valor x representa um valor lógico desconhecido

Tipos de dados estruturais

A linguagem Verilog suporta tipos de dados estruturais. Os dois tipos mais comuns de dados estruturais são o *wire* e o *reg*. O *wire* é usado para criar lógica combinacional. O tipo *reg* guarda valores até que outro valor é colocado sobre este, não é necessariamente um registo (FF). As declarações dos sinais *wire* e *reg* encontram-se dentro do módulo, mas fora de qualquer bloco *inicial* ou *always*. Oestado inicial de um*reg* é x e do *wire*éz.

Tipos de dados comportamentais

Os tipos de dados *integer* e *real* são tipos de dados convenientes para serem usados em contagem dentro de blocos de código comportamental. Se se deseja sintetizar um determinado código comportamental, deve se então evitar este tipo de dados porque eles sintetizam circuitos muito grandes.

O tipo de dados *time* pode conter um valor especial de simulação chamado de tempo de simulação, que é extraído da função do sistema $time. A informação do tempo pode ser usada para ajudar no *debug* das simulações.

Definição de constantes

Em Verilog, a definição de constantes permite a adição de parâmetros de largura. Por exemplo:

8’h79 → representa 8 bits em hexadecimal com o valor de 79.
4’b1101 →representa 4 bits em binário com o valor de 1101.

Como se pode ver no exemplo a sintaxe usada é: $< largura\ em\ bits >' < letra\ da\ base >< número >$.

Estruturas de dados

A linguagem Verilog suporta três tipos de estruturas de dados, *arrays*, vectores, e memorias. Os *arrays* são usados para armazenar vários objetos do mesmo tipo. Os vectores são usados para representar barramentos multi-bit. As memórias são *arrays* de vectores que são acedidos de maneira semelhante às memórias de *hardware*.

Operadores

Na tabela 2.2 aparece uma seleção de operadores da linguagem Verilog que são muito importantes.

Tabela 2.2 – Operados da linguagem Verilog

Tipo	Símbolo	Operação realizada
Operador binário	~	Bitwise NOT
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR
	~^ ou^~	Bitwise XNOR
Lógico	!	NOT
	&&	AND
	||	OR
Redução	&	Redução AND
	~&	Redução NAND
	|	Redução OR
	~|	Redução NOR
	^	Redução XOR
	~^ ou^~	Redução XNOR
Aritmético	+	Adição
	-	Subtração

	-	Complemento 2
	*	Multiplicação
	/	Divisão
	**	Exponencial
Relacional	>	Maior do que
	<	Menor do que
	>=	Maior ou igual do que
	<=	Menor ou igual do que
	==	Igualdade lógica
	!=	Diferença lógica
	===	Igualdade lógica 4-state
	!==	Diferença lógica 4-state
Shift	>>	Deslocamento lógico à direita
	<<	Deslocamento lógico à esquerda
	>>>	Deslocamento à direita aritmético
	<<<	Deslocamento à esquerda aritmético
Concatenação	{ }	Concatenação
Replicação	{n{m}}	Replica o valor m n vezes
Condicional	? :	Condicional

***Constructs* sintetizáveis**

Em Verilog existem uma série de declarações que não tem semelhanças com o *hardware* real, o que provoca que grande parte da linguagem não possa ser usada para a descrição de hardware, como por exemplo $display.

De seguida está apresentado um exemplo de código que tem mapeamento direto para os *gates* (portas) reais.

```
always@(a or b or sel)
begin
if(sel)
out= a;
else
out= b;
end
```

Fork/join

Para criar processos paralelos em Verilog é usadoo par *fork/join*. Todos os blocos ou declarações feitas entre os pares *fork/join* iniciam a execução em simultâneo. A execução continua após o join após a conclusão da execução da maior declaração ou bloco entre o fork e o join. Na figura 2.3 aparece o funcionamento geral do par *fork/join.*

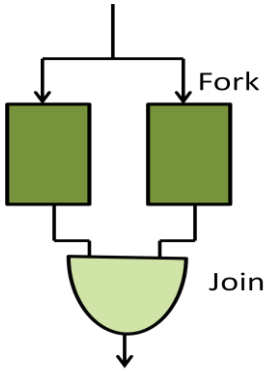


Figura 2.3 – Visão geral do funcionamento do par *fork/join.*

Um exemplo da implementação em Verilog é a seguinte:

```
initial
fork
   $write(“A”);
   $write(“B”);
begin
#1;
$write(“C”);
end
join
```

No exemplo em cima, é possivel que sejam imprimidas as seguintes sequencias de “ABC” ou “BAC”. A ordem de simulação entre o primeiro $write e o segundo $write depende da implementação da simulação. Isto ilustra um dos grandes problemas do Verilog. Pode se ter race conditions onde a ordem de execução não garante os resultados.

Race conditions

A linguagem Verilog não garante sempre a ordem de execução, então se duas expressões são programas para executarem ao mesmo tempo, e se o modo de execução não se encontra determinado, ocorre uma condição de corrida.

Uma race condition ocorre quando dois ou mais declarações que estão programadas para executar na mesma simulação, provocaria resultados diferentes quando a ordem de execução da instrução é alterada.

Tarefas do sistema

A linguagem Verilog disponibiliza tarefas de sistema para que possam lidar com Entradas/Saídas, e com várias funções de medição de *design*. De forma a distinguir as tarefas de sistema das tarefas de utilizador e funções, é usado o prefixo $, alguns exemplos são apresentados a seguir:

$random – retorna um valor aleatório.
$time – valor do tempo da simulação actual
$display – Escreve num ficheiro uma linha seguida por uma nova linha automaticamente.
$monitor – Imprime todas as variáveis listadas quando qualquer valor muda.
$write – Escreve no ecrãuma linha sem introduzir uma nova linha.

Program Language Interface (PLI)

A linguagem Verilog disponibiliza a PLI, que dá ao utilizador uma API (*Application Program Interface*) para Verilog. Essencialmente a PLI é um mecanismo que chama funções da linguagem C a partir do código Verilog. A PLI permite que um código Verilog possa cooperar com outros programas escritos em C, tais como equipamentos de teste, *debuggers*, e assim por adiante.

A linguagem Verilog é uma boa linguagem baixo nível. Modelos estruturais são de *design* fácil e o código RTL comportamental é muito bom. A sintaxe do Verilog é regular e de fácil de lembrar. É uma linguagem HDL de fácil uso e aprendizagem. Contudo o Verilog carece de tipos de dados definidos pelo utilizador e não tem separação interface-objeto do modelo entidade-arquitetura do VHDL.

2.1.3. SystemVerilog

A linguagem SystemVerilog [19] [20] é uma combinação da linguagem de descrição de *hardware* e linguagem de verificação de *hardware*, baseando-se em Verilog.

O principal incentivo do SystemVerilog está na verificação de projetos electrónicos, onde a maior parte

das funcionalidades de verificação estão baseadas na linguagem OpenVera[1] doada pela empresa Synopsys.

Em 2005, o SystemVerilog foi adotado como padrão do IEEE.

As funções podem ser dívidas em duas funções distintas:

A linguagem SystemVerilog para projetos RTL são uma extensão do Verilog-2005, e todas as características presentes neste estão disponíveis em SystemVerilog.

A linguagem SystemVerilog para verificação usa técnicas extensivas de programação orientada a objetos que se encontram mais intimamente relacionadas com C++.

Como referido anteriormente, o SystemVerilog baseia-se em Verilog, como tal engloba a maioria dos aspetos de projeto de sistema e *testbench* que o Verilog oferece, tal como os aspetos que vemos na figura 2.4.

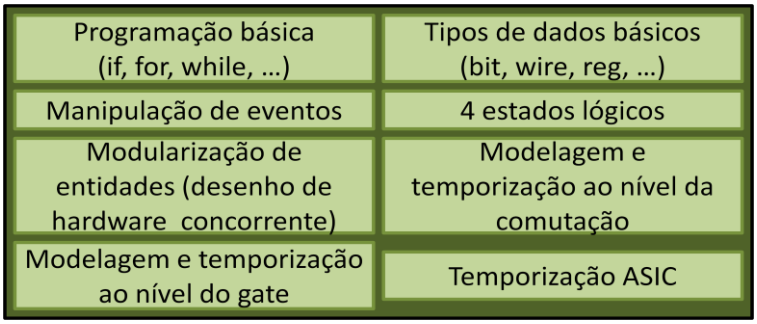


Figura 2. 4 – Verilog 95: Linguagem de projeto e testbench (bloco verde)

Em SystemVerilog também foram adicionadas algumas funções avançadas de VHDL, como podemos ver na figura 2.5.

[1] OpenVera é uma linguagem de verificação de *hardware* aberta para a criação de testbench, traz interoperabilidade e foi desenvolvido pela Synopsys.

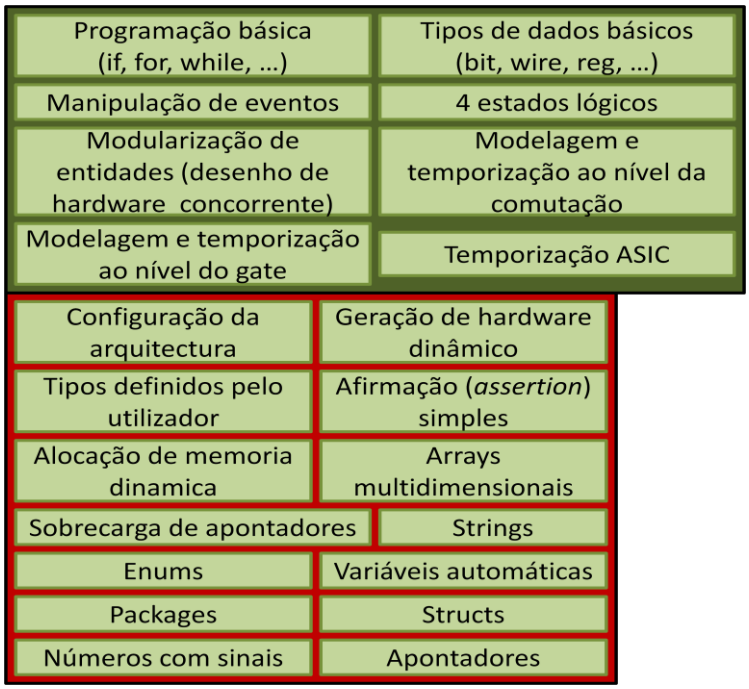


Figura 2.5 - VHDL adiciona ao SystemVerilog tipos de dados de alto nível e funcionalidades de gestão (bloco vermelho).

Várias das funcionalidades acima referidas são comuns a linguagem C, como por exemplo, apontadores, arrays multidimensionais, tipos de dados básicos, programação básica, etc. Funcionalidades extra da linguagem C foram acrescentadas como se pode ver na figura 2.6.

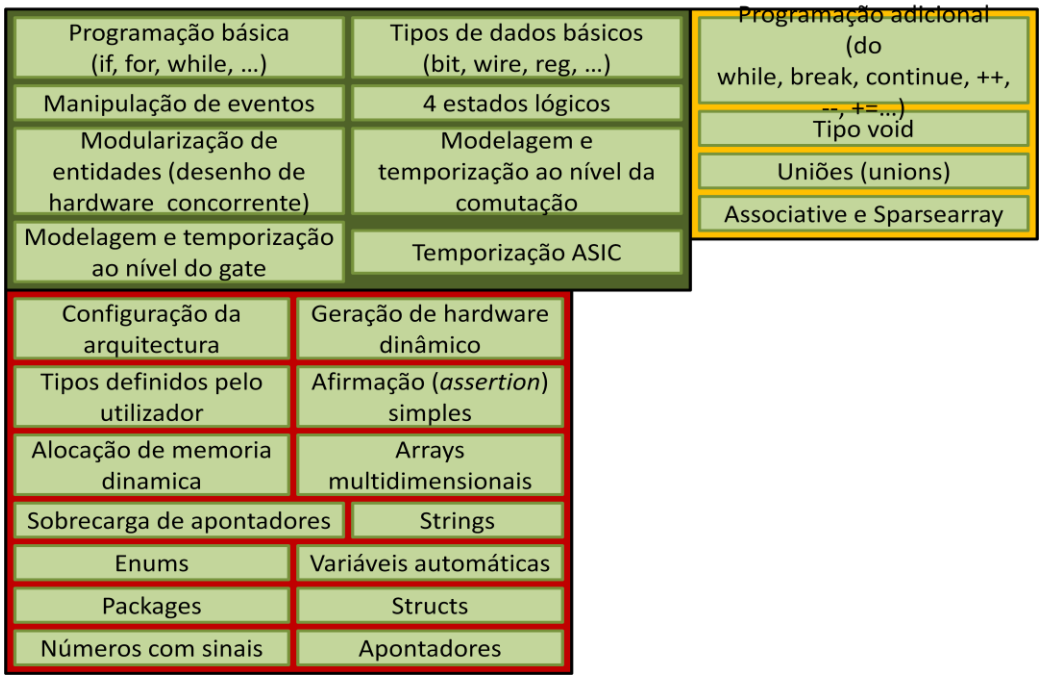


Figura 2. 6 – A linguagem C trazrecursos de programaçãoextra ao SystemVerilog (bloco amarelo).

Uma vez que a linguagem SystemVerilog é uma combinação da linguagem de descrição de *hardware* (HDL) e da linguagem de verificação de *hardware* (HVL), faz com que procure funcionalidade que não se encontravam disponíveis em Verilog, VHDL e C, como podemos ver na figura 2.7. O SystemVerilog oferece recursos avançadosde verificação ede modelação.

Figura 2.7 - Linguagem de SystemVerilog completa (bloco azul, vermelho, amarelo e verde).

A linguagem SystemVerilog como referido, traz um maior nível de abstração para o *design* e verificação. Como tal fornece um ambiente de verificação completa, e uma melhoria nas características de modelação de *hardware*, que melhoram a produtividade do design RTL e simplifica o processo de *design*. De seguida será feita uma breve abordagem as principais funcionalidades introduzidas pelo SystemVerilog, que não se encontram presentes no Verilog, fazendo uma pequena comparação entre eles.

Interfaces

A linguagem SystemVerilog adiciona interfaces, ondeas ligações entre modelos são agrupados, reduzindo a redundância de declarações *portname* entre módulos conectados, as definições deconexões são independentes dos módulos, e os protocolos e verificações podem fazer parte da interface. Na figura 2.8 retrata o que foi referido anteriormente

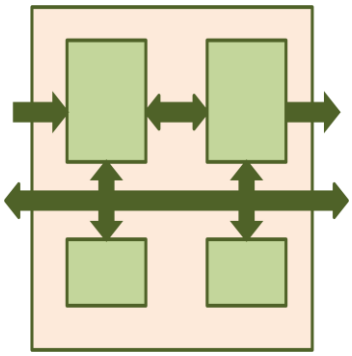


Figura 2.8 – Esquema geral das conexões dos módulos em SystemVerilog.

A linguagem Verilog, por sua vez, conecta módulos usando os portos dos mesmos, ou seja, requer conhecimento detalhado das conexões para interligar os módulos. Como não usa interfaces, caso se pretenda fazer alterações de *design* é bastante complexo, além disso exige que em cada módulo os portos estejam duplicados, como se pode ver na figura 2.9.

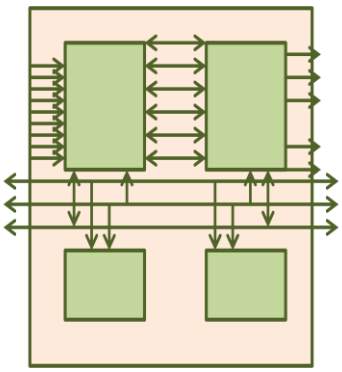


Figura 2. 9 – Esquema geral das conexões dos módulos em Verilog.

Extensões de unidade de tempo (fs ps ns ms s step)

A linguagem SystemVerilog acrescenta unidades de tempo, que podem ser especificadas como parte do valor de tempo, eliminando assim qualquer ambiguidade conforme o que o atraso representa, conforme se pode ver no exemplo seguinte, é apresentado um oscilador de 10 nanosegundos.

```
forever #5ns clock = ~clock;
```

As unidades de tempo também podem ser especificadas como parte do módulo, onde os arquivos podem ser compilados em qualquer ordem.

```
module my_chip(…);
timeunit 1ns;
timeprecision 10ps;
...
```

Por sua vez, em Verilog, as unidades de tempo fazem parte da simulação, e são declaradas com a directiva de compilador `timescale, e ao contrário do que acontece no SystemVerilog, existem dependência da ordem em que os ficheiros são compilados. No exemplo que se segue aparece o mesmo oscilador, mas neste caso não se especifica a unidade de tempo, não ficando claro se se trata de um oscilador de 10 nanossegundos, ou de outra unidade qualquer.

forever #5 clock = ~clock;

Melhorias nas atribuições dos valores literais

O SystemVerilog aumenta as atribuições de valores literais, para tal é acrescentada uma nova sintaxe, `<value> que estende a todos os bits o valor desejado, que pode ser 0, 1, X, x, Z, z.

data_bus = `1; →Preenche todos os bits do data_bus a um.
data_bus = `z; →Preenche todos os bits do data_bus com zzzzzzzzzzzzzzzz (hex).

As atribuições em Verilog-1995, estendem com o valor 0 quando atribuído a um vetor *unsigned*, e ao valor lógico Z até 32 bits, quando passar dos 32 bits estende com zeros.

reg [63:0] data_bus;
data_bus = 64'hFFFFFFFFFFFFFFFF; → Preenche com 1s.
data_bus = `bz; →Preenche com 00000000zzzzzzzz(hex).

Quando criado o Verilog-2001 as atribuições estenderam-se a lógica Z para todos os bits (não compatível com Verilog-1995)

data_bus = `bz; →Preenche com zzzzzzzzzzzzzzzz (hex)

O SystemVerilog também adiciona *time literals*, que dá a opção de especificar os atrasos com unidades explícitas, funciona da seguinte maneira:

```
#5ns b <= !b;
#1ps $display("%b", b);
```

E adiciona *array literals*, onde o funcionamento é semelhante ao C, mas com permissão do uso do operador de replicação ({{}}).

intn[1:2][1:3] = {{0,1,2},{3{4}}}

Melhoria dos módulos de conexões de porto

A linguagem SystemVerilog remove a maioria das restrições sobre as conexões de portos, como tal permite que qualquer tipo de dados (números reais, arrays, estruturas) possam conectar-se em ambos os lados dos portos. Por sua vez, a linguagem Verilog restringe os tipos de dados que podem ser conectados aos portos do módulo, uma vez que no lado de transmissor, apenas podem ser conectados *nets*, *regs* ou inteiros, e no lado de receção apenas tipos de redes, o que torna bastante difícil obter o tipo correto.

Tipos de dados

A linguagem SystemVerilog acrescenta tipos de dados abstratos, tais como tipos 2-state (*int, shorting, longint, bit, byte*) e tipos especiais (*void, shortreal*). Os tipos 2-state, são baseados ao bit e podem ter os valores 0 e 1, por sua vez os tipos 4-state podem ter os valores de 0, 1, Z e X. Graças ao tipo de dados 2-state, o SystemVerilog permite misturar *2-state* e *4-state* no mesmo *design*, o que produz resultados determinísticos, ou seja todas as ferramentas utilizam a mesma semântica nos tipos de dados.

O SystemVerilog adiciona ***Multidimensional packed arrays***, queunifica e estende a noção de Verilog sobre registos e memorias.
O Verilog clássico apenas permite uma dimensão a ser declarada à esquerda do nome da variável.

reg [7:0]vetor; → Vector de oito bits.
reg[3:0]matriz[0:63]; → Vector de 64 posições, sendo cada uma de 4bits.

SystemVerilog permite qualquer número de dimensões “packed”

logic [3:0][7:0]matriz[0:10]; → 10 entradas de matrizes 4x8. Configurando assim uma matriz tridimensional

Tipos enumerados

A linguagem SystemVerilog acrescenta tipos enumerados (***enum***), que têm uma sintaxe similar ao C. ***Enum*** permite que a quantidades numéricas possam ser atribuídos nomes significativos. Este tipo de dados é adequado e útil para representação de valores de estado, *opcodes*, etc.

```
typedef enum reg [2:0] {
        RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW
} color_t;
color_t my_color = GREEN;
```

O Verilog não tem tipos enumerados, como tal todos os sinais devem ser declarados e inicializados com um valor.

```
parameter START = 3'b000;
parameter WAIT = 3'b001;
parameter LOAD = 3'b010;
```

Tipos definidos pelo utilizador

O SystemVerilog acrescenta tipos definidos pelo utilizador (*typedef*). *Typedef* permite aos utilizadores criar nomes para definir tipos de dados que serão usados com frequência, podem ser muito convenientes quando se constroem *arrays* complicados de definir.

```
typedef int unsigned uint;
Uint a, b;                    →  inteiros unsigned

typedef enum {FALSE=1'b0, TRUE=1'b1} boolean;
boolean ready;                →  o sinal "ready" pode ser  true ou false
```

O Verilog não tem tipos de dados definidos pelo utilizador, apenas tipos de dados incorporados (wire, reg, integer, real, ...) podem ser usados.

Estruturas e Associações

A linguagem SystemVerilog acrescenta estruturas e associações que funcionam como na linguagem C. Uma estrutura é um conjunto de dados que se encontraram inter-relacionados, isto é os membros de uma estrutura podem ser definidos separadamente e mesmo assim é possível

trabalhar com todos eles. O tipo de dados *union* (associação) é muito semelhante ao tipo estrutura, mas apenas um membro é válido num dado momento de tempo.
A linguagem SystemVerilog acrescenta várias funcionalidades únicas às estruturas e associações, inclui o atributo *packed* e *tagged*, onde o atributo *tagged* permite que seja possível acompanhar em tempo de execução de qual membro da união que está atualmente em uso, já o atributo *packed*, permite "empacotar" uma estrutura na memória sem intervalos entre os domínios de bits, isto é útil para o acesso rápido de dados durante a simulação.

```
typedef union {
        int i;
        integer j;
}data_word;
```

```
typedef struct packed {
        int x, y;
        } Point;
Point p;
```

O Verilog não tem uma maneira conveniente de agrupar diversos sinais juntos, para tal deve-se usar módulos como pseudo-estruturas.

Extensões aos Operadores da linguagem Verilog

O SystemVerilog adiciona operadores de incremento e decremento (++ e --), operadores de atribuição (+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, <<<=, >>>=)

```
for (i = 0; i <= 255; i++)
…
```

A linguagem Verilog não tem os operadores de C, incremento e decremento

```
for (i = 0; i <= 255; i = i + 1)
…
```

Melhorias nos ciclos *for*

O SystemVerilog aumenta os ciclosfor, permitindo que a variável do ciclo possa ser declarada dentro do ciclo for, e podem haver múltiplas atribuições iniciais e de etapas, como podemos ver a seguir.

```
for (int i=1,shortint count=0; i*count < 125; i++, count+=3)...
```

Nos ciclos *for* em Verilog, a variável de controlo do ciclo é declarada antes do ciclo, e apenas pode haver uma atribuição inicial e de etapa.

```
integer i;
for (i = 0; i <= 255; i = i + 1)...
```

Melhorias para reduzir a ambiguidade dos modelos

A linguagem Verilog é muito utilizada para a síntese de código RTL, é muito fácil escrever código Verilog que simule corretamente, mas que mesmo assim produza um *design* incorreto, uma vez que usa procedimentos de modelação de uso geral.A linguagem SystemVerilog introduz 3 novas abordagens de *always*: always_comb, always_ff e always_latch.

O *always_comb* é um modelo usado para descrever a lógica combinacional. Implicitamente cria uma lista de sensibilidade analisando as variáveis que são lidas no processo, funciona exatamente como o "always@" da linguagem Verilog-2001. Por sua vez, o *always_ff* é um bloco que se destina a inferir na lógica síncrona, e o *always_latch* é um bloco que se destina a inferir em uma *level-sensitive latch*.

Outro erro muito comum em códigos RTL escritos na linguagem Verilog é o uso indevido do *parallel_case* e *full_case,* os problemas aparecem porque são eles são ignorados pelos simuladores que os tratam como comentários, mas eles são utilizador para síntese direta. O SystemVerilog resolver este problema com a ajuda do*priority* e *unique*.O *priority* e *unique* são aplicáveis em declarações *if* e *case.* O *unique*reforça a integridade e singularidade das condições, ou seja, é possível realizar em tempo de execução a verificação de somente uma entrada. Com o *unique t*odos os valores tem que pertencer ao *case*ou senão tem que colocar o valor *default*. O *priority* impõe um conjunto de verificações menos rigorosas, verificando apenas se pelo menos um ramo da condição é usado.

Melhorias para simplificar *Netlist*

A linguagem SystemVerilog acrescenta abreviaturas de forma a simplificar as conexões de portos. Usando a sintaxe .name é adicionada a capacidade de instanciar implicitamente portas, se o nome da instancia do porto e o tamanho coincidir com a conexão do nome da variável do porto e do tamanho, por sua vez o .* faz a conexão dos portos universais.

Desta forma, é eliminada a necessidade de listar o nome do porto duas vezes quando o nome do porto e o nome do sinal são os mesmos, como se pode verificar no exemplo que se segue.

```
module my_chip (input wire clock, reset);
wire [63:0] a, b, c, d;
alu u1 (.reset, .a, .b, .c, .d, .ck(clock));
alu u2 (.*,.ck(clock));
```

→ .name reduz a digitação quando o sinal e o porto têm o mesmo nome

→.* conecta automaticamente os sinais e os portos que têm o mesmo nome

A linguagem Verilog permite conexões de *netlist* usando módulo de ordem dos portos (simples, mas propenso a erros), e módulo de nomes de portos (repetitivo).

Melhorias na proteção IP

Os módulos SystemVerilog podem ser declarados dentro de outros módulos. Os módulos recebidos são locais para o seu módulo principal, e as outras partes do *design*não podem ver os módulos locais.

No exemplo abaixo é possível verificar que existem 3 módulos globais, o Top, my_stuff e ip_core, e como tal podem ser instanciados em qualquer lugar, por sua vez o modulo alu é um módulo local dentro do modulo global ip_core, e apenas pode ser usado dentro desse módulo.

```
module Top;
        my_stuff i1 (clock);
        ip_core    i2 (clock);
endmodule
module my_stuff (input bit clock);
...
endmodule
module ip_core    (input bit clock);
        module alu();
                ...
        endmodule
endmodule
```

Em Verilog, os nomes dos módulos são sempre globais. O sub-módulo num bloco IP é global e pode ser referenciado em qualquer lugar do projeto.

Assertions

A linguagem SystemVerilog adiciona sintaxe e semântica de *assertions*, onde uma declaração especifica o comportamento de um sistema. O principal uso das *assertions* é de validar o

comportamento de projetos, podendo ainda serem usadas para proporcionar cobertura funcional e para gerar estímulos de entrada para a validação.

O SystemVerilog permite *assertions* para transmitir informações ao *testbench*, e permite que o *testbench* que reaja a condição de *assertions* sem a necessidade do uso de uma API separada de qualquer tipo.

A construção das informações das *assertions*ocorre dentro da linguagem, sem a necessidade de módulos de *pragmas* especiais ou chamadas ao PLI (*Programing Language Interface*). As *assertions* em SystemVerilog são construídas a partir de sequências (*sequences*) e propriedades (*properties*), onde as propriedades são um super conjunto de sequências, e qualquer sequência pode ser usada como se fosse uma propriedade. Uma sequência consiste em expressões booleanas aumentadas com operadores temporais.

```
sequence request_check;
        request ##[1:3] grant ##1 !request ##1 !grant;
endsequence
always @(posedge clock)
        if (State == FETCH)
        assert property request_check;
```

A linguagem Verilog não fornece *assertions.* A verificação do modelo deve ser feita por *hard-coded logic.* Através de bibliotecas de afirmações, como as bibliotecas OVL, são adicionadas verificações de afirmações na forma de instâncias de módulo. Também é possível adicionar afirmações através de ferramentas que usando propriedades "pragmáticas", ou então ferramentas que usam PLI.

Melhorias aos blocos dos *testbench*

A linguagem SystemVerilog adiciona um "*program block*" que contém um ambiente completo para *testbench*. O *Program block* fornece um ponto de entrada para a execução de *testbench*, para tal cria um objecto que encapsula os dados, tarefas e funções no projeto, e fornece um contexto sintático que provoca com que os eventos sejam executados em uma fase reactiva.

A linguagem Verilog utiliza módulos para modelar o *testbench.* Os módulos são destinados a modelos de *hardware*, e não têm semântica especial para evitar *race conditions* com o *design.*

Melhorias nos ciclos baseados em verificação de sincronismo

A linguagem SystemVerilog introduz o uso de *clocking blocks* para que fosse possível resolver os problemas de especificar os requisitos de tempo e sincronização de um projeto nos *testbenches*. Os*clocking blocks*são conjuntos de sinais sincronizados num relógio particular, que separa os detalhes relacionados com o tempo dos elementos estruturais, funcionais e procedimento do *testbench*. Os *clocking blocks*apenas podem ser declarados dentro de módulos, interfaces ou programas.

```
Clocking bus @(posedge clk);
        default input #2ns output #1ns;
        input enable, full;
        inout data;
        output empty;
        input reset;
endclocking
```

O Verilog não tem uma especial semântica de *clock* para verificação. Como tal o engenheiro de verificação deve definir relógios de teste extras para evitar condições de corrida com o relógio de *design*.

Extensões dos tipos de dados: Classes orientadas a objetos

A linguagem SystemVerilog introduz conceitos de orientação a objeto através da abstração classe com abstração de dados, tendo como intenção primária o uso em verificação. As classes permitem que os objetos sejam criados dinamicamente, eliminados, atribuídos, e acedidos através de manipulação de objetos.

Em System Verilog as classes suportam um modelo único de herança e têm série de semelhanças a linguagem C++, uma vez que suportam polimorfismo, encapsulamento público, local ou privado e novos objetos criados e inicializados usando *new*.

```
classPacket ;
        bit [3:0] command;
        bit [39:0] address;
        bit [4:0] master_id;
        integer time_requested;
        integer time_issued;
        integer status;
        taskclean();
                command = 4'h0;
                address = 40'h0;
```

```
            master_id = 5'b0;
        endtask
        task issue_request(int delay);
        ...
        endtask
endclass
```

Melhorias de sincronização: *Mailboxes* e semáforos

O SystemVerilog acrescenta um conjunto de mecanismos de sincronização e de comunicação que podem ser criando e alterados de forma dinâmica, semáforos e *mailboxes.* Os semáforos em SystemVerilog podem ser usados para sincronização e exclusão mútua de recursos partilhados, isto é os semáforos representam um recipiente com um número fixo de chaves, implementam métodos utilizados para verificar as chaves *in* e out e osprocessos podem verificar uma ou mais chaves, e devolvê-los mais tarde, e se não houver chaves suficientes disponíveis, a execução do processo pára e espera pelas chaves antes de continuar. Os *mailboxes*, em SystemVerilog podem ser usados como um canal de comunicação entre processos, ou seja representam um FIFO para troca de mensagens entre processos e constroem métodos que permitem a adição de uma mensagem ou recuperar uma mensagem, e se nenhuma mensagem estiver disponível, o processo pode esperar até que uma mensagem seja adicionada, ou continuar e verificar novamente mais tarde.

Melhorias nos valores aleatórios

A linguagem SystemVerilog adiciona uma melhoria nos números aleatórios, tais como:

- **$urandom**, que gera números aleatórios de 32 bits *unsigned.*
- **$urandom_range**, quegera números aleatórios dentro de um intervalo específico de 32 bits *unsigned.*
- **rand**, constrói a classe para a criação de números aleatórios distribuídos, isto é, um valor aleatório pode ocorrer mais de uma vez antes que todos os valores possíveis numa escala limitada tenham ocorrido.
- **randc**, constrói a classe para a criação de números aleatórios cíclicos, ou seja, todos os valores possíveis dentro da faixa restrita ocorrerão uma vez e apenas uma vez, e então uma nova sequência aleatória começará.
- Valores aleatórios controlados restringidos com construção de métodos de classe.

Na linguagem Verilog o uso do **$random** retorna um número aleatório de 32 bits signed, mas não há forma de restringir os valores aleatórios retornados.

Melhorias nos processoes *Fork-Join*

A linguagem SystemVerilog expandiu a *construct fork-join*, para tal adicionou processos paralelos dinâmicos usando *fork-join_any* e *fork-join_none*, presentes na figura 2.10.

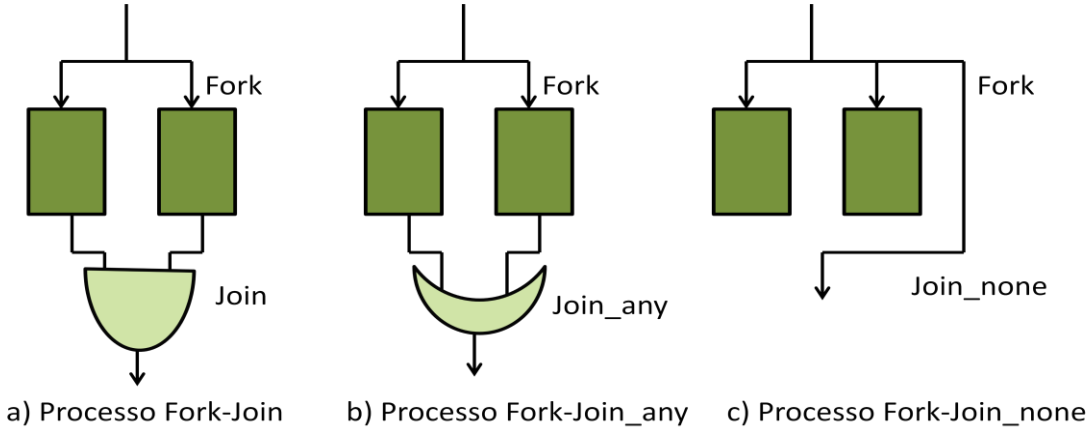


Figura 2.10 (a, b, c) – Constructs fork-join

A *constructfork-join*permite a criação deprocessos simultâneosa partir de cada uma de suasdeclaraçõesparalelas, como se pode verificar na figura 2.10 (a). O fork-join tem a seguinte sintaxe:

```
type_of_block @(sensitivity_list)
fork
        statement1;
        statement2;
        ----
join
```

O bloco *fork-join* provoca sempre que o processo execute a instrução *fork* para bloquear até que todos os processos *forked* terminem. Com a adição do *join_any* e *join_none*, o SystemVerilog fornece três opções para especificar quando o processo paicontinua a execução.
No *fork-join_any*, o processo pai bloqueia até que todos os processos gerados pelo *fork* estejam completos (figura 2.10 (b)).

No *fork-join_none*, o processo pai continua a executar simultaneamente com todos os outros processos gerados pelo *fork* (figura 2.10 (c)). Os processos gerados não iniciam a execução até que a *thread* pai execute uma instrução de bloqueio.

O Verilog suporta processos paralelos usando *fork-join*, onde todos os processos paralelos devem terminar a execução antes de continuar

Direct Program Interface (DPI)

A linguagem SystemVerilog adiciona uma interface de programação direta (DPI) que é basicamente uma interface entre SystemVerilog e uma linguagem de programação externa, nomeadamente a linguagem C. O DPI permite que um *designer* possa facilmente chamar funções de C a partir de SystemVerilog e exportar funções do SystemVerilog de modo a que possa ser chamado pela linguagem C.
O DPI traz vantagens que consistem em permitir ao utilizador reutilizar código C já existente e também não requer o conhecimento da linguagem de programação de interface do Verilog (PLI), ou da interface procedimental do Verilog (VPI), além disso fornece uma maneira alternativa bastante fácil de chamar algumas funções PLI ou VPI, mas não todas.

A linguagem Verilog usa a interface de linguagem de programação (PLI) para permitir que o código Verilog possa chamar o código C, uma vez que o PLI permite ao Verilog cooperar com outros programas escritos na linguagem C, tais como equipamentos de teste, simuladores de conjuntos de instruções de um microcontrolador, depuradores, etc. O PLI tem capacidades poderosas: atravessando hierarquias, controle de simulação, sincronização com o tempo de simulação, etc, mas é de difícil aprendizagem, por causa de todas as capacidades, o que o torna muito complexo para muitos tipos de aplicações.

Cobertura (Coverage)

O *SystemVerilog* adiciona *Coverage* que quando é aplicado a linguagens de verificação de *hardware*, refere-se à recolha de dados estatísticos com base em eventos de amostragem dentro da simulação. Ou seja, o *coverage* é usado para determinar quando o DUT (dispositivo sob teste) foi exposto a uma suficiente variedade de estímulos, para garantirque o DUT está a funcionar corretamente.

2.1.4 Outras Técnicas Atuais de Extensão de Linguagem

Existem outras técnicas atuais de extensão de linguagem que foram desenvolvidas para estender funcionalidades às linguagens trazendo novas vantagens resolvendo possíveis problemas que uma linguagem possa ter.

Bons exemplos de paradigmas de extensão de linguagem são o *Template Metaprogramming*, *Model-Driven Design*, *Generative Design* e *Transaction Level Modeling*(TLM). Por exemplo, a linguagem SystemC obedece a TLM, fornecendo a capacidade de modelar *hardware* e *software* simultaneamente.

A TLM [22] supõe um novo enfoque ao problema da modelagem e desenho de sistemas hardware/software. A peça chave da modelação com TLM é a utilização de transações com abstração da comunicação entre diferentes módulos num sistema. O conceito de modelação baseado em transações está presente em muitas línguas de programação de alto nível, mas a sua história está ligada a linguagem SystemC. Uma transação em TLM é, dependendo da especificação do sistema, uma transferência de dados entre dois módulos ou um evento de sincronização. A TLM da suporte para uma metodologia e fluxo de desenho que tem como principal vantagema de oferecer um modelo único de referência para múltiplas equipas de trabalho que podem assim desenvolver a sua atividade de maneira concorrente.

A TLM tem como intuito oferecer uma solução de compromisso entre um modelo de alto nível – bom para capturar a essencial do sistema – e um modelo para simulação que seja rápida e ao mesmo tempo proporcione dados fiáveis – bom para orientar decisões que proporcionem a melhor implementação do sistema tendo em conta as restrições definidas.

2.2. Introdução aos protocolos de encriptação

Nesta secção será feita uma breve abordagem a vários protocolos de encriptação. Para tal é importante entender uma série de conceitos relacionados com a criptografia, assim como o porque da criação de protocolos de encriptação.

A criptografia é um estudo pertencente à criptologia, e consistem em transformar qualquer tipo de dados da sua forma original para uma forma ilegível que é apenas conhecida pelo destinatário detentor de uma chave secreta, desta forma, impossibilitando a leitura dos dados por alguém não autorizado. Ou seja, consiste em aplicar processos matemáticos aos dados que desejamos que sejam secretos (a que se dá o nome de cifragem) de forma a obter dados ilegíveis que posteriormente irão circular por um meio inseguro ate chegarem ao destinatário. Para que sejam obtidos os dados originais é necessário reverter o processo anterior (decifragem). Desta forma, os dados só serão lidos pelo destinatário.

Os objetivos criptográficos são os seguintes: Confidencialidade, autenticação, integridade e não repúdio.

A confidencialidade é um serviço usado para proteger a informação daqueles que não tem autorização para aceder a ela. A autenticação é um serviço relacionado à identificação, esta função é aplicada a ambas as entidades e àprópria informação em si. Duas partes que queiram comunicação devem primeiro identificar-se uma à outra, e a informação entregue através do canal deve ser autenticada quanto à origem, data de origem, conteúdo de dados, hora de envio, etc. A integridade dos dados é um serviço que trata das alterações não autorizadas nos dados. Para que a integridade dos dados seja garantida, é necessário ter a capacidade de detetar a manipulação dos dados por pessoas não autorizadas, por manipulação de dados pode entender-se como inserção, exclusão e substituição. Do ponto de vista da comunicação não segura, o não repúdio é visto como fornecendo autenticação e integridade, isto é não pode negar de quem vem a mensagem e o seu conteúdo.

O elemento central de um sistema de criptografia é a cifra, que efetua as transformações entre os dados limpos e o criptograma. As cifras modernas são famílias de funções em que cujo comportamento depende de um parâmetro numérico denominado por chave. Na figura 2.11podemos ver o funcionamento dos algoritmos/cifras baseados em chaves.

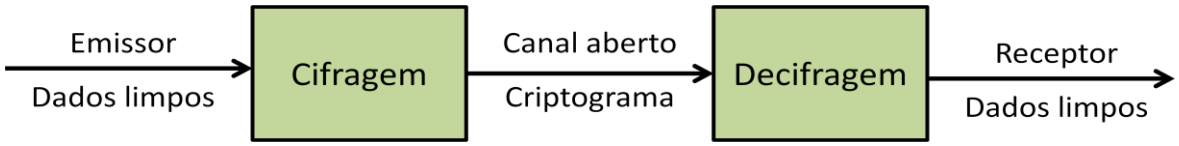


Figura 2.11 – Funcionamento de algoritmos/cifras baseados em chaves.

É importante distinguir entre duas classes de algoritmos criptográficos, os algoritmos de cifra simétrica ou chave secreta e os algoritmos de cifra assimétrica ou chave pública. As cifras simétricas caracterizam-se por usarem chaves criptográficas iguais tanto para cifrar como para decifrar os dados. Os interlocutores usam a mesma chave, e esta representa um segredo partilhado entre eles e apenas por eles, de forma a manterem uma ligação confidencial. Nas cifras assimétricas, as chaves de cifragem e decifragem são diferentes. Apenas a chave de decifragem necessita de ser secreta, e apenas o recetor precisa de a conhecer. Um intruso pode conhecer a chave de cifragem, sem que isso comprometa a segurança da cifra.

Apesar de que aparentemente as cifras assimétricas pareçam mais seguras que as cifras simétricas, devido ao facto de que nas cifras simétricas ser mais complexo a gestão e distribuição das chaves, o que acontece na realidade é que as cifras assimétricas não são um substituto para as cifras simétricas, umas vez que as primeiras são muito mais lentas que as segundaspara o mesmo nível de segurança. Por esses motivos todos os algoritmos de criptografia que serão abordados neste documento, serão todos algoritmos de cifras simétricas. Dentro das cifras simétricas ainda podemos fazer distinção entre cifras sequenciais e cifras por blocos.

As cifras sequenciais operam sobre *streams* de dados, um bit ou um byte de cada vez, combinando-o com um *stream* de chaves geradas internamente. O esquema de decifragem é idêntico ao de cifragem. A sequência de chaves tem de ser reproduzida exatamente para recuperar o texto limpo. A segurança da cifra reside na dificuldade de prever a sequência de valores gerados sem se saber a chave da cifra, isto é, reside totalmente no gerador de chaves.

As cifras por blocos operam sobre blocos de tamanho fixo (tradicionalmente era de 64 bits, mas atualmente de 128 ou 256 bits). Associados à função de cifragem, são necessárias unidades de partição e *padding*[2], para produzir blocos de tamanho apropriado. Na decifragem, a função inversa da função de cifragem é aplicada aos blocos do criptograma. Os processos de *padding* e partição são revertidos no final.

[2] O *padding* de uma mensagem surge da necessidade das cifras por blocos requerem um texto limpo com o tamanho múltiplo do tamanho do bloco. Assim, o *padding* consiste em preencher os espaços que faltam. Ou seja, o bloco de texto limpo é completado com bytes aleatórios concatenados no fim do texto limpo.

2.2.1. DES (*Data Encryption Standard*)

O Data Encryption Standard (DES) é uma cifra simétrica por blocos, desenvolvido pela IBM sendo depois adotado como *standard* pelo NIST (*National Institute of Standards and Technology*).

O DES cifra blocos de 64 bits em blocos de 64 bits, e para tal usa uma chave de 64 bits (56 bits de dados e 8 bits de paridade). Desta forma, para cifrar uma mensagem, escolhe-se um chave e divide-se a mensagem em blocos de 64 bits, sendo cada bloco cifrado separadamente. O algoritmo DES consiste na aplicação sucessiva (16 vezes) de uma operação de cifragem ao bloco de dados, chamado de *round*.

A chave de cifra é processada para produzir 16 novas chaves de 48 bits, uma chave para cada *round*, este processo é chamado de *key Schedule*, e consiste na permutação da chave de cifrar.

Cada *round*do DES é construído com base na chamadaRede de Feistel, que consiste em partir cada bloco em dois sub-blocos (E e D) (ver figura 2.2)de 32 bits cada, onde em um bloco é aplicada uma transformação e o outro é preservado, o que provoca que sejam necessários dois *rounds* para cifrar todos os bits do bloco. Os sub-blocos são ainda trocados para que o *round*seguinte afete agora o sub-bloco que ficou inalterado. O sub-bloco que é cifrado passa por uma função de Feistel (F), onde é permutado, combinado com a chave de *round*, e passado por uma função não linear, como é possível verificar na figura 2.12.

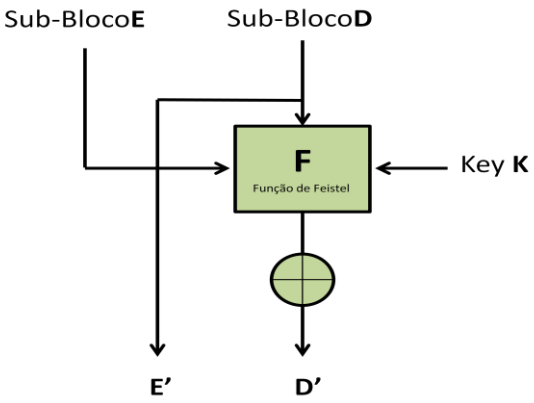


Figura 2. 12– Rede de Feistel Clássica

Devido ao facto de cada *round* no DES ser construído com base num circuito de Feistel, permite implementações muito eficientes, onde o mecanismo de cifragem e decifragem são idênticos, mas visto ser uma operação inversa, a ordem de aplicação das chaves de round também é inversa, e por esse motivo o algoritmo conclui com a inversão da permutação inicial.

A implementação do DES envolve permutação(*P-boxes*), substituição (*S-boxes*), e *key schedule*.
Em [1], apoia o uso do algoritmo DES num coprocessador de encriptação ASIC (*Aplication-Specific Integrated Circuit*) de baixo consumo de energia para nós de rede de sensores, devido ao facto do DES não incluir operações complexas de matemática e de *power-hungry* como divisão, multiplicação e adição.
O DES atualmentecomeça a evidenciar algumas das suas fragilidades, principalmente devido ao pequeno tamanho da chave, o que torna o DES relativamente de pouca utilidade para comunicações inter-satélite uma vez que a suachave é de apenas 56 bits e o seu nível de complexidadeé muito alto [7].

2.2.2. 3DES (*Triple Data Encryption Standard*)

O Triple Data Encryption Standard é um algoritmo de criptografia simétrica por blocos, baseado no algoritmo DES.

O 3DES usa 3 chaves, K1, K2 e K3, de 64 bits cada (8 bits dos 64 são de paridade) o que faz com que o tamanho máximo da chave seja de 192 bits, e o tamanho do bloco de texto original é de 64 bits.
O funcionamento do algoritmo, pode ser visto da seguinte forma:

$$Texto\ cifrado = CK3(DK2(CK1(Texto\ Original)))$$

Onde as funções C e D são de cifragem e decifragem respetivamente. Onde a primeira chave (K1) serve para cifrar o texto original, seguido da segunda chave (K2) para decifrar os dados. É de notar que DK2 não retorna o texto original, e por ultimo a terceira chave (K3) voltar a cifrar, o que provoca com que o 3DES seja mais lento mas ofereça maior segurança comparativamente ao DES original.
Em [2] apresenta-se a implementação do algoritmo 3DES em sistemas embebidos para terminais PoS (*Point of Sale*), que tem como principal requisito que o micro controlador usado seja de baixo custo com desempenho satisfatório para o utilizador. Com o algoritmo 3DES é possível cumprir os requisitos acima referidos num micro controlador limitado tanto a nível de capacidade de processamento como em memória disponível.

2.2.3. IDEA (*International Data Encryption Algorithm*)

O *International Data Encryption Algorithm* (IDEA) [3] é um algoritmo simétrico por blocos desenvolvido em 1990 por L. Massey e Xueija. Durante alguns anos (nas décadas de 90) foi a melhor cifra por blocos no mercado. Mas mesmo assim não substituiu o DES pelo seguinte motivo: é patenteado (o proprietário da patente é o ASCOM) então é necessária uma licença.

No IDEA o tamanho do bloco é de 64 bits, e a chave é de 128 bits, e o mesmo algoritmo é utilizado tanto para cifragem como para decifragem, permitindo implementações eficientes.

O algoritmo baseia-se na partição do bloco de 64 bits em quatro sub-blocos de 16 bits cada. O algoritmo IDEA passa por oito *rounds*, onde em cada *round* ocorre uma aplicação sucessiva de operações algébricas (XOR, adição módulo 2^{16} e multiplicação módulo $2^{16} + 1$) aos sub-blocos e aos bits da chave.

Cada *round*dispõe de uma fase difusora(cada sub-bloco é processado independentemente) e uma fase misturadora (onde os sub-blocos se interferem mutuamente).

Particularmente adaptado para realizações em *software*(operações em palavras de 16 bit, onde só a multiplicação necessita de código específico). A decifragem só difere no programador de chaves (são produzidas as chaves inversas pela ordem adequada).

Os campos típicos onde são usados os algoritmos IDEA, como podemos ver em [3] é um campo mais direcionados para o *software*, tal como referido anteriormente, uma vez que é um algoritmo facilmente embutido em qualquer software de criptografia, e devido ao facto de o IDEA ser um algoritmo seguro e que permite uma proteção eficaz dos dados transmitidos e armazenados contra o acesso não autorizado por terceiros. É referido o uso do IDEA em dados de áudio e vídeo para televisão por cabo, televisão por assinatura, videoconferência e VoIP, assim como dados financeiros confidenciais e comerciais, e-mail através de rede publicas, ligações de transmissão via modem, *router* ou ATM *link*, tecnologia GSM, e cartões inteligentes, entre outros.

2.2.4. RC4 (*Rivest Cipher 4*)

RC4 [4] é uma cifra simétrica sequencial desenhada por Ron Rivest em 1987 para RSA *Security*. É uma cifra sequencial com tamanho da chave variável e com operações orientadas ao byte. O algoritmo RC4 é usado nos standards SSL/TLS (*Secure Sockets Layer/Transport Layer Security*) que foram definidos para a comunicação entre *browsers* e servidores. O algoritmo também é usado no protocolo WEP (*Wired Equivalent Privacy*) e no protocolo WPA (*WiFi Protected Access*). O RC4 foi mantido como um segredo comercial pela RSA *Security*.

O funcionamento do algoritmo RC4 é bastante simples e é baseado na utilização de uma permutação aleatória. Uma chave de tamanho variável que pode ir de 1 a 256 bytes (8 a 2048 bits) é usada para inicializar um estado do vector *S* a 256 bytes, com elementos de *S*[0], *S*[1], …, *S*[255]. Em todos os momentos, *S* contém uma permutação de todos os números de 8 bits de 0 a 255. Para a cifragem e decifragem é gerado um byte k a partir do S selecionando uma das 255 entradas de uma forma sistemática. Como cada valor de *k* gerado, as entradas em *S* são mais uma vez permutadas.

2.2.5. AES (*Advanced Encryption Standard*)

O AES é um algoritmo de criptografia de blocos simétrico, também conhecido por *Rijndael*, foi escolhido como resultado de um concurso, e foi desenvolvido por investigadores de uma universidade belga. O AES foi anunciado pela NIST(*National Institute of Standards and Technology*) em 26 de Novembro de 2001 e tornou-se padrão em 26 de Maio de 2002.
OAESé um algoritmode chave simétrica, em que tanto o emissor como recetorusamuma única chavepara cifrar e decifrar, o comprimento do bloco de dados é de 128 bits, eo comprimentos da chave é variável, podendo ser de 128, 192 ou 256 bits. Éum algoritmo iterativo e o número de *rounds*, *Nr*, pode ser de 10, 12 ou 14 com o respectivo comprimento de chavede 128, 192ou 256bits.
Cada *round*noAES, com exceção do último *round* passa por quatro etapas: *SubBytes*, *ShiftRows*, *MixColumns*e*AddRoundKey*.

- SubBytes: A transformação SubByte é uma substituição não linear que opera em cada byte da matriz de estado independentemente. Esta é a operação mais complexa do AES e garante que a não-linearidade é introduzida durante a cifragem. Estes elementos são referidos como *SBoxes*.

- ShiftRows: Nesta transformação, as linhas da matriz de estado são alteradas (deslocadaspara a direita) ciclicamente sendo C1, C2 e C3 o deslocamento que se deve aplicar nas linhasda matriz de estado onde a primeira linha não é alterada, a segunda linha é deslocada em“*C1”* bytes, a terceira linha é deslocada em “*C2”*bytes e a quarta linha é deslocada em “*C3”* bytes. Estes parâmetros *C1*, *C2* e *C3* dependemdo número de bytes (*Nb)* da coluna.

- MixColumns: cada coluna é processada através de multiplicação de vetores em campo binário. É a principal operação de difusão da AES, e não é calculado no *round* final da AES.

- AddRoundKey: Esta transformação consiste somente em aplicar uma operação lógica "ou exclusivo" (XOR) na matriz de estado, usando a matriz proveniente da função de escalonamento de chaves.

No AES o resultado das diversas operações intermediárias (4 etapas de cada *round*) realizadas é colocado na denominada "matriz de estado" ou "Estado". Uma matriz de estado podeser visualizada comouma matriz retangular com quatrolinhas, o número de colunasno estado é indicado por*Nb*eé igual aocomprimento do bloco em bitsdividido por32.Para umbloco de dados128bits (16 bytes)o valor de*Nb*é 4,portanto,o estadoé tratado como uma matriz4 x 4 e cadaelemento da matriz representa umbyte.

Tabela 2.3- Tabela de combinações

	Tamanho da chave (Nk words)	**Tamanho do bloco (Nb words)**	**Número de rounds (Nr)**
AES – 128	4	4	10
AES – 192	6	4	12
AES – 256	8	4	14

O algoritmo AES foi projetado para teras seguintes características [21]:

- Resistência contratodos os tipos de ataques conhecidos;
- Velocidade ecompactação de um códigoem umaampla gama de plataformas;
- Simplicidade de *design*.

O processo de decifragem é o inverso do de cifragem.
A codificação AES foi precisamente concebida para resistir aos ataques de descodificação mais avançados, como os métodos de análise de tempo e análise de potência. Para proporcionar ao utilizador ainda mais benefícios, a codificação AES, quando idealmente administrada necessita de pouca memória para codificar e descodificar.

Vários artigos apoiam o uso do algoritmo AES. Em [4]e [5] apoiam o uso do algoritmo AES em satélites pequenos de observação do planeta Terra, devido a grande procura para a proteção dos dados transmitidos desde os satélites para a solo, e como tal fazem referência ao facto de o algoritmo AES revelar-se como a opção preferida da indústria aeroespacial, incluindo satélites.

Em [7] fazem referência ao uso de chaves criptográficas para comunicações inter-satélites, fazendo uma avaliação breve de vários protocolos de criptografia simétrica.

2.3. Introdução a técnicas de tolerância a falhas

Nesta secção serão apresentadas técnicas e conceitos de tolerância a falhas devido à suscetibilidade a falhas de execução por parte dos sistemas digitais, e à necessidade acrescida deste tipo de técnicas para maior fiabilidade dos sistemas.

2.3.1. Tolerância a falhas

Os sistemas digitais são suscetíveis a falhas de execução que podem ser causadas por diversos fatores. Quando um sistema exige alta fiabilidade deve ser construído usando técnicas de tolerância a falhas. Esses tipos de técnicas garantem um melhor funcionamento dos sistemas quando ocorrem falhas, e são baseadas em redundância, exigindo algoritmos especiais. A tolerância a falhas não dispensa técnicas de prevenção e remoção de falhas.

As técnicas de tolerância a falhas têm várias abordagens, onde a mais comum encontra-se dividida em quatro fases [8].

A primeira fase é a de deteção de erros. Uma falha para que seja detetada deve primeiro manifestar-se como um erro, isto porque se isto não acontecer ela está latente e não pode ser detetada. Isto é, uma falha pode permanecer no sistema toda a sua vida sem nunca se manifestar, ou seja, nunca leva o sistema a um estado erróneo.

A segunda fase é a de confinamento e avaliação. Como foi referido anteriormente uma falha pode ser latente, e após a ocorrência de uma falha até que se manifeste como um erro, pode haver um espalhamento de dados inválidos. Os sistemas em si não provem o confinamento, e como tal devem ser previstas e implementadas restrições ao fluxo de informação e estabelecer interfaces de verificação para deteção de erros.

A terceira fase é a de recuperação de erros. Esta ocorre após a deteção dos mesmos e abrange a troca do estado incorreto, para um estado livre de falhas.

A quarta e última fase é a de tratamento de falhas, e consiste em localizar a origem do erro (falha), localizar a falha de forma precisa, corrigir a falha, e recuperação do sistema.

2.3.2. Falhas mascaradas

Uma falha nem sempre se manifesta como erro, então o sistema não entra num estado erróneo, mas pode haver propagação de dados inválidos devido a uma falha, nestes casos as falhas são mascaradas de forma a que garantam respostas corretas no sistema mesmo havendo falhas nos mesmos. Alguns mecanismos típicos na implementação de falhas mascaradas são: replicação de componentes, ECC (códigos de correção de erros), blocos de recuperação, etc.

2.3.3. Redundância

Um sistema tolerante a falhas tem de mascarar falhas nos seus componentes, de forma a que não afetem o seu comportamento. A redundância tem um forte impacto no sistema, tanto em questões de custo, como de desempenho, área, ou energia consumida. Pode usar-se redundância tanto para deteção de falhas, como para mascarar as mesma. Conforme o caso pretendido, o grau ou tipo de redundância pode mudar, o que torna necessário distinguirmos três tipos de redundância.

A **redundância espacial** corresponde à existência de *hardware* repetido para realizar uma determinada função, isto é, uma ação crítica é executada por componentes replicados, o que não provoca atrasos nos processos de deteção e recuperação de erros. No caso de uma falha não ser mascarada, esta pode provocar que o estado do sistema seja erróneo, e então é necessário que seja tratado para que não gere defeito e para que a deteção de erro seja utilizada. Após a deteção do erro, é importante evitar que danos que tenham sido provocados pelo erro não se propaguem pelo sistema e só depois é que se deve realizar o processamento para recuperação do sistema do estado de erro ou então compensar o erro através de uso de redundância.

A **redundância temporal** repete a computação no tempo, ou seja, é feita uma ação de correção após a deteção da falha, o que provoca um aumento de tempo necessário, mas trás a vantagem de evitar custos de *hardware*. Este tipo de redundância é pouco aplicada a sistemas críticos, uma vez que o processo de recuperação pode corromper as especificações temporais. A redundância temporal é usualmente usada para deteção de falhas transitórias, uma vez que resultados diferentes em computação indicam uma alta probabilidade de haver uma falha transitória, e deteção de falhas permanentes, através da cifragem e decifragem de dados. Os dados antes da cifragem e após a decifragem devem ser iguais, se não for esse o caso uma falha permanente é manifestada.

A **redundância de informação** pode ser vista como o uso de dados provenientes de diversas fontes, ou como o uso de *backup* de dados, ou inclusive como o uso de códigos de deteção de erros. Para tal são adicionados aos bits de dados bits ou sinais extras que não contém informação útil que servem para a deteção de erros ou para mascarar falhas. Alguns exemplos são os de códigos de paridade, *checksums*, códigos de duplicação, códigos cíclicos, todos eles usados para deteção de erros. Quando se deseja mascarar falhas através do uso de redundância de informação é necessário a utilização de códigos de correção de erros, como ECC (*error correction code*).

2.3.4. TMR (Triple Modular Redundancy)

O TMR é uma das técnicas mais conhecidas de tolerância a falhas [9] [10]. Os componentes de *hardware* são triplicados e a saída dos mesmos é comparada, isto é, o esquema de mitigação usa três circuitos lógicos idênticos (*register*) que executam a mesma tarefa em paralelo e as saídas correspondentes são comparadas por um *voter*,,como se pode ver na figura 2.13. Este*voter* realiza uma função muito simples, e consiste em 3 portas lógicas NAND e um OR cuja correção é fácil de verificar, a votação é feita pela maioria resultando numa só saída, o que significa que se um dos sistemas falhar, os outros dois estando corretos mascaram o erro. Se houver falhas em mais que um componente o *voter*torna-se vulnerável.

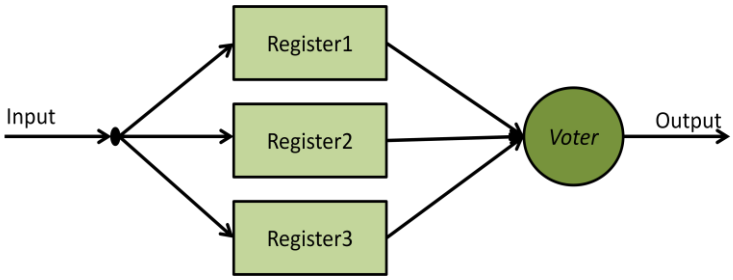


Figura 2.13 – Esquema básico do TMR.

Em [9] apoiam o uso de TMR, como técnica de tolerância a falha, para detetar e reduzir os efeitos de SETs (*Single-Event Transients*) de forma a diminuir a suscetibilidade dos sistemas digitais a falhas.

Em [10] também apoiam o uso de TMR mas neste caso para proteção das funcionalidades dos FPGAs contra SEUs (*Single-Event Upsets*) e ainda é feita uma descrição da implementação do código TMR, assim como os resultados das medições de desempenho, e uma breve discussão das limitações que o método apresenta.

2.3.5. Códigos Reed-Solomon

Os códigos *Reed-Solomon* (RS) são um subconjunto dos blocos lineares BCH [1, 2], são código cíclicos e podem ser construídos de forma a corrigir múltiplos erros. Os códigos cíclicos são uma subclasse dos *block codes standard* de deteção e correção de erros que protegem a informação contra erros nos dados transmitidos sobre um canal de comunicações. Os códigos *Reed-Solomon* pertencem a categoria FEC (*Forward Error Correction*), isto é corrige os dados alterados no recetor e para tal utiliza uns bits adicionais que permitem a recuperação *à posteriori*. Os códigos *Reed-Solomon* são muito usados em telecomunicações, tanto em meios de armazenamento como compactação, e transmissões distantes, devido à capacidade de correção de um elevado número de erros dentro da mesma palavra.

A especificação de um código *Reed-Solomon* é RS (n, k, t), onde o *n* representa o número de símbolos de saída, o *k* é o número de símbolos de entrada, e o *t* é a capacidade de correção do código. Desta forma, este código pode corrigir até *t* símbolos errados numa palavra código onde $2t = n - k$, como mostra a figura 2.14.

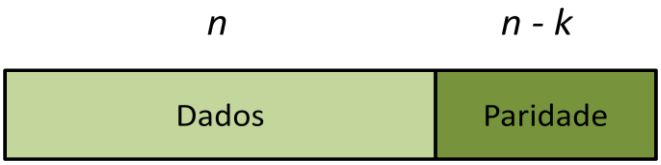


Figura 2.14 -Estrutura de um dado codificado com códigos Reed-Solomon.

A implementação dos códigos de *Reed-Solomon* (RS), não tem em consideração características de design como custo de área e desempenho quando se trata de uma implementação em *hardware*, visto que a sua execução é normalmente feita em *software*.

Em [11] propõem o uso dos códigos Reed-Solomon combinados com códigos Hamming para a proteção de memórias SRAM contra falhas múltiplas, mostrando uma eficácia de 100% para falhas duplas.

2.3.6. Códigos de Hamming (*Hamming codes*)

Códigos de Hamming são códigos binários de deteçãoede correção de erro. Os códigos de Hamming podem detetar um ou dois bits de erro, e apenas corrigir um bit de erro que possa ocorrer quando os dados binários são transmitidos a partir de um dispositivo para outro. O princípio fundamental que é abraçado pelos códigos de Hamming é a paridade, como tal os códigos de Hamming adicionam bits de paridade ao bloco de dados, de forma a que, caso ocorram erros de transmissão, seja possível deteta-los e/ou corrigi-los [11].

Os códigos de Hamming satisfazem a seguinte equação $d + p + 1 \leq 2^p$, onde *d* corresponde ao número de bits de dados e *p* ao numero de bits de paridade. Seguindo essa equação os códigos de Hamming podem corrigir todos os bits de um erro em *d* bit palavras e detetar erros de bit duplo quando um bit de paridade geral é usado (SECDED - *Single Error Correction Double Error Detection*). A implementação destes códigos é composta por um bloco combinacional responsável por codificar os dados, por bits extras na palavra que indica a paridade e por outro bloco combinacional responsável por decifrar os dados. A utilização mais usual destes códigos apresenta-se em aplicações de telecomunicações, computação, assim como aplicações que incluem a compressão de dados e *turbo codes.*

Código SEC (*Single Error Correction*)

A deteção de erros de um bit único numa mensagem recebida é possível através do uso de paridade.

Para que haja correção de erros é necessário haver mais informação, uma vez que a posição do bit corrompido deve ser identifica para que possa ser corrigido. Para que a correção seja possível não pode existir apenas um bit de paridade, uma vez que um erro qualquer de bit, em qualquer posição produz exatamente a mesma informação, ou seja, erro. Se forem incluídos bits extra à mensagem e, se esses bits forem dispostos de tal forma que diferentes bits corrompidos possam produzir diferentes resultados de erro, então os bits corrompidos podem ser identificados.

Em [12]é proposta a implementação dos códigos Hamming (SEC) para uma estrutura de *hardware* reconfigurável para correção de bits de erro numa linha de comunicação

Código SEC-DED (*Single Error Correction Double Error Detection*)

Para muitas aplicações um código de correção de um erro de bit único pode ser insatisfatório, porque ele aceita todos os blocos recebidos. O código Hamming pode converter-se de um código SEC em SEC-DED por adição de um bit de verificação, que é um bit de paridade em todos os bits da palavra do código SEC. Nem sempre é percetível que está presente um SEC-DED, porque se existir um erro no bit de paridade este pode comporta-se como um código SEC, como se vai verificar.

Tabela 2.4- Adição do bit paridade para criar um código Hamming SECDED

Erros	Paridade	Síndrome	Conclusões
0	Par	0	Não existem erros
1	Impar	0 ≠ 0	Erro no bit de paridade Síndrome indica que existe um erro
2	Par	≠ 0	Erro duplo (não corrigível)

Como é possível ver na tabela 2.4, se não existirem erros, a paridade é par (por paridade entendem-se os *n* bits de uma palavra código recebida), e a síndrome de *n-1* bits do bloco é de 0. No caso de ocorrer um erro, a paridade é impar, se o erro ocorrer em um bit de paridade, a síndrome é 0, mas se o erro ocorrer noutro bit qualquer, a síndrome será diferente de zero e indicará em que bit se encontra o erro. Se houver 2 erros, o bit paridade será par, e a síndrome será diferente de zero, mas deve-se ter em conta que se um dos erros for no bit de paridade, o código funciona como um SEC.

3. Materiais e Métodos

Nestecapítulo é feita uma descrição dos materiais e métodos utilizados durante toda a elaboração desta dissertação. A base deste trabalho como anteriormente referido, consiste no estudo de diferentes HDLs, e identificar os pontos onde o nível de abstração pode ser aumentando. Para tal, é necessário a implementação utilizando HDLs de um *case study*, que consiste na implementação, em *hardware*, de um algoritmo de encriptação eficiente para aplicações aeroespaciais com tolerância a falhas. A implementação, das técnicas de tolerância a falhas é essencial, uma vez que se deseja ter um alto desempenho a baixo custo de área, o que torna a análise das melhores técnicas de tolerância a falhas de suma importância.

Após um estudo intensivo foi decidido o desenvolvimento do algoritmo de encriptação em AES usando a linguagem de descrição Verilog, o software utilizado para a implementação das funcionalidades é o Xilinx ISE Design Suite 13.2 e Hardware COTS (*Comercial of-the-shelf*) usado para sintetizar o sistema é o FPGA Xilinx Virtex 5. De forma a garantir a fiabilidade e segurança dos sistemas foram implementadas duas técnicas de tolerância a falhas sobre o AES, de forma a ser possível analisar o desempenho de cada uma delas, as técnicas implementadas foram o *Triple Modular Redundancy* e os códigos Hamming. Como extensão à linguagem SystemVerilog/Verilog, foi desenvolvido um pré processador para lidar com registos TMR desenvolvido em linguagem C, compatível com os sistemas operativos Linux e Windows. Ao algoritmo de encriptação AES foi aplicada a nova extensão desenvolvida de forma a ser possível comparar as diferentes implementações em termos de algumas métricas selecionadas, incluindo custo de engenharia no desenvolvimento e legibilidade.

3.1. Extensões à linguagem SystemVerilog

A implementação de sistemas com TMR usando HDLs limitam a produtividade dos projetistas, como tal foi necessário definir *constructs* de modo a aumentar o nível de abstração da linguagem SystemVerilog. Desta forma, foi desenvolvido um SystemVerilog/Verilog pré-processador para lidar com registos TMR, que consiste num código em C que abre um ficheiro de entrada.ve analisa-o, depois escreve um ficheiro de saída equivalente.No ficheiro de entrada sempre que se deseja utilizar TMR basta usada um registo “reg_tmr”, o SystemVeriog/Verilog pré-processador interpreta o ficheiro de entrada sempre que encontra um “reg_tmr” este é substituido por três registos idênticos seguidos de um *voter*. Como é possível verificar no exemplo seguinte:

```
reg_tmr [7:0] acc;       // registo usado no ficheiro .v de entrada.
```

é substituído por

```
reg [7:0]  acc1, acc2, acc3;
wire [7:0]       acc;

voter #(8)voter_acc(acc1,acc2,acc3,acc);
```

A variável declarada acc é adicionada a uma tabela de símbolosde forma a que todas a posteriores escritas sejam substituídas pela escrita de registos TMR, isto é,

```
acc<= b;
```

ésubstituído por,

```
acc1 <= b;
acc2 <= b;
acc3 <= b;
```

Na figura 3.1 seguinte é apresentado um exemplo de um ficheiro .v de entrada que usa reg_tmr.

```
module TMR_AND(Clk, a, b, d);

input             Clk;
input   [7:0]  a;
input  [7:0]   b;
output [7:0]   d;

wire [7:0] c;

assign c = a & b;

reg_tmr          [7:0]    acc;

always@(posedge Clk)
begin
        acc <= c;
end

assign d = acc;

endmodule
```

Figura 3.1 – Código Verilog que faz uso das novas metodologias.

O ficheiro de entrada representado na figura 3.1 é lido e analisado pelo pré-processador de registos TMR. É executado um *token* de cada vez através do ficheiro de entrada até encontrar um EOF (*End Of File*), ao longo do processo vai escrevendo para o ficheiro de saída.

Na figura3.2 seguinte é representado o ficheiro processado de saída _tmr.v equivalente.

```
module TMR_AND(Clk,a,b,d);

input   Clk;
input   [7:0]   a;
input   [7:0]   b;
output [7:0] d;
wire [7:0] c;

assign c = a & b;

reg     [7:0]   acc1, acc2, acc3;
wire [7:0] acc;

voter #(8) voter_acc(acc,acc1,acc2,acc3);

always@(posedge Clk)
begin
acc1 <= c;
acc2 <= c;
acc3 <= c;
end
assign d = acc;
endmodule
```

Figura 3. 2 – Código Verilog gerado pelo pré processador de registos TMR.

Usando estas novas metodologias é possível implementar sistemas com TMR de uma forma simples e de fácil perceção por terceiros, uma vez que o código RTL desenvolvido torna-se mais legível e compacto.

O código em C do pré processador de registos TMR encontra-se mais adiante na secção de anexos (Anexo A).

3.2. Caso de estudo

Neste subcapítulo é apresentado o caso de estudo desenvolvido, cujo principal foco são aplicações aviónicas e aeroespaciais, onde é necessário assegurar fiabilidade e seguranças dos sistemas. Como tal encontra-se neste capitulo descrito o desenvolvimento de todo o caso de estudo tendo em conta todo o estudo realizado sobre os diversos protocolos de encriptação, técnicas de tolerância a falhas e as HDLs.

3.2.1. AES (Advanced Encryption Standard)

Após uma análise e avaliação dos vários protocolos de encriptação, decidiu-se que o algoritmo AES é o mais adequado, uma vez queutiliza operações algébricas de forma perspicaz: a segurança é conseguida através de operações complexas, mas de implementação muito eficiente tanto em *hardware* como em *software*, o que torna o AES altamente adequado para comunicações inter-satélite, uma vez que é capaz de usaras chaves criptográficasde 128, 192e 256bits paracifrar e decifrar dadosem blocos de128 bits, oferecem simplicidade, flexibilidade de implementação e alto *throughput*.

OAES como referido anteriormente é um algoritmode chave simétrica, em que tantoo emissor como orecetor usamuma única chavepara cifrar e decifrar. O comprimento do bloco de dados definido é de128 bits,e oscomprimentos de chavepodem ser de 128, 192 ou 256 bits.O número total de*rounds*, Nr, pode ser de 10, 12 ou 14 dependendo dos respetivos comprimentos de chavede 128, 192 ou 256 bits. Cada *round*noAES, exceto o último,consiste de quatrotransformações:ShiftRows, SubBytes, MixColumnseAddRoundKey, uma de permutação e três de substituição, que serão explicadas em mais detalhe mais adiante.No AES o resultado das diversas operações intermediárias realizadas é colocado em um lugar denominado “matriz de estado” ou apenas Estado (*State*), que podeser visualizado comouma matriz retangularcom quatrolinhas.O número de colunasno estadoé denotado porNbeé igual aocomprimento do blocoem bitsdividido por32.Para umbloco de dados de 128bits (16bytes)o valor deNbé de 4,portanto, oEstado étratado como uma matriz4 x4 e cadaelemento da matrizrepresenta umByte.

A estrutura geral do algoritmo AES, é apresentada na figura 3.3. Na figura pode-se ver que a entrada tanto para o algoritmo de cifragem como para o de decifragem é um bloco único de 128 bits, onde esse bloco é copiado para a matriz de estado. A cifra começa pela etapa de AddRoundKey, por ser a única etapa que faz uso da chave, depois são realizados 10 *rounds*, uma

vez que na nossa implementação optamos usar uma chave de 128 bits, sendo o *round* final diferente dos anteriores, pois neste não ocorre a etapa MixColumns. Após o *round* final, a matriz de estado é copiada para a saída e desta maneira o processo termina. Estas operaçõessão representadasna figura3.4.

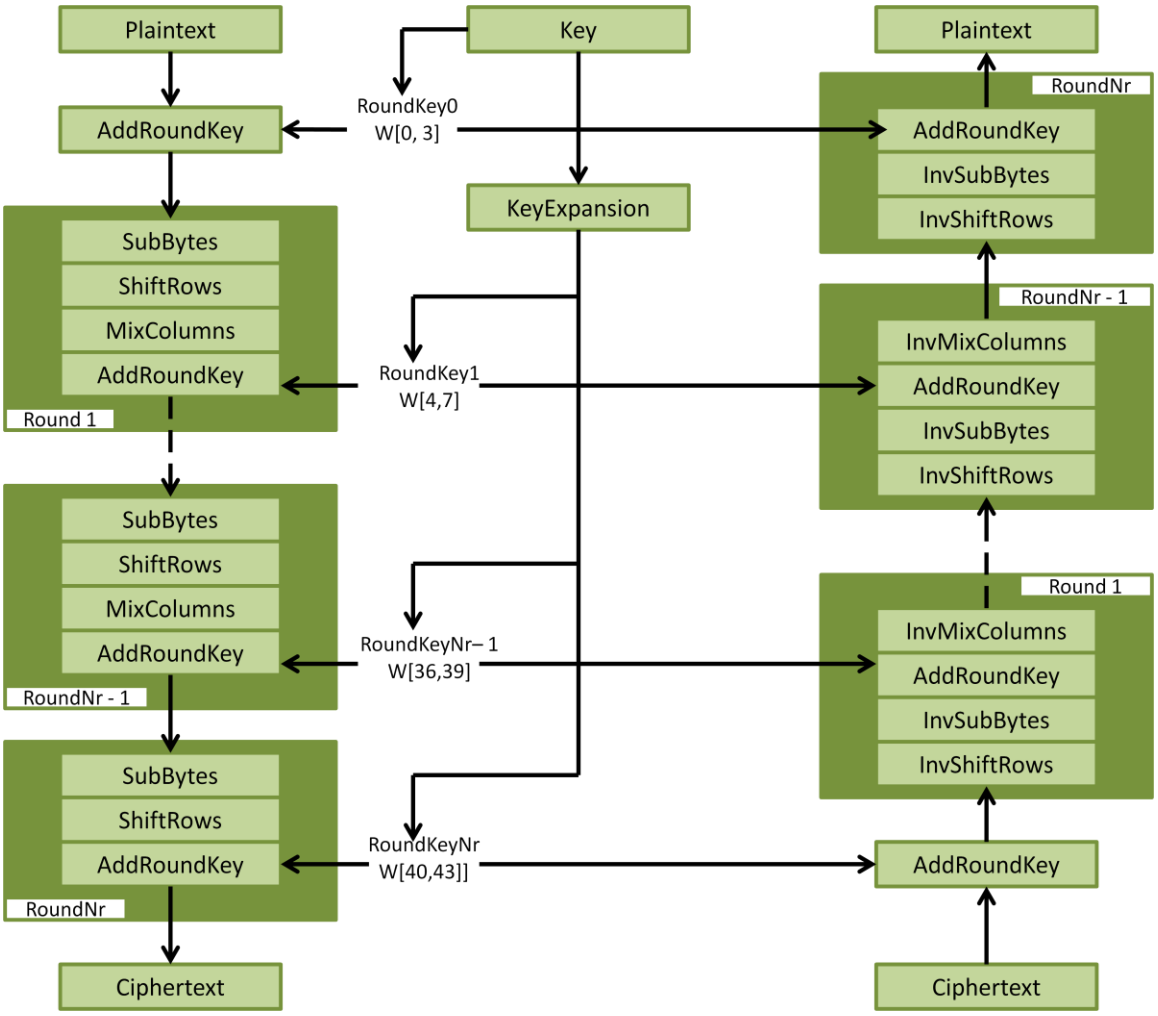


Figura 3.3 – Estrutura geral do algoritmo AES

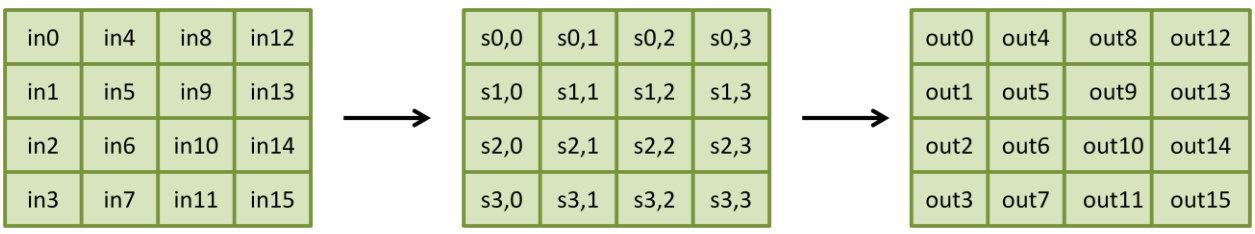


Figura 3.4 – Estrutura dos dados AES (Input, state, output)

Da mesma forma, a chave de 128 bitsé representada comouma matriz quadrada debytes, onde esta chaveé então expandida num *array* de *key schedule words*, onde cada palavraé de quatrobytese o *key schedule* totalé de 44palavras, como se pode observar na figura 3.5.

É importante referir quea disposição dosbytesdentro da matrizépor coluna. Por exemplo, os primeiros quatro bytesde entrada dotexto simplesde 128 bits no cifrador ocupam aprimeira coluna damatriz, os segundos quatro bytesocupam asegunda coluna eassim consecutivamente.Da mesma forma,os quatro primeirosbytesda chaveexpandida (*expanded key*),que formamuma palavra, ocupam a primeira colunada matrizw.

Figura 3.5 – Estrutura dos dados AES (key e expanded key)

Na figura 3.6 é mostrada a estrutura de um round de cifra completa.

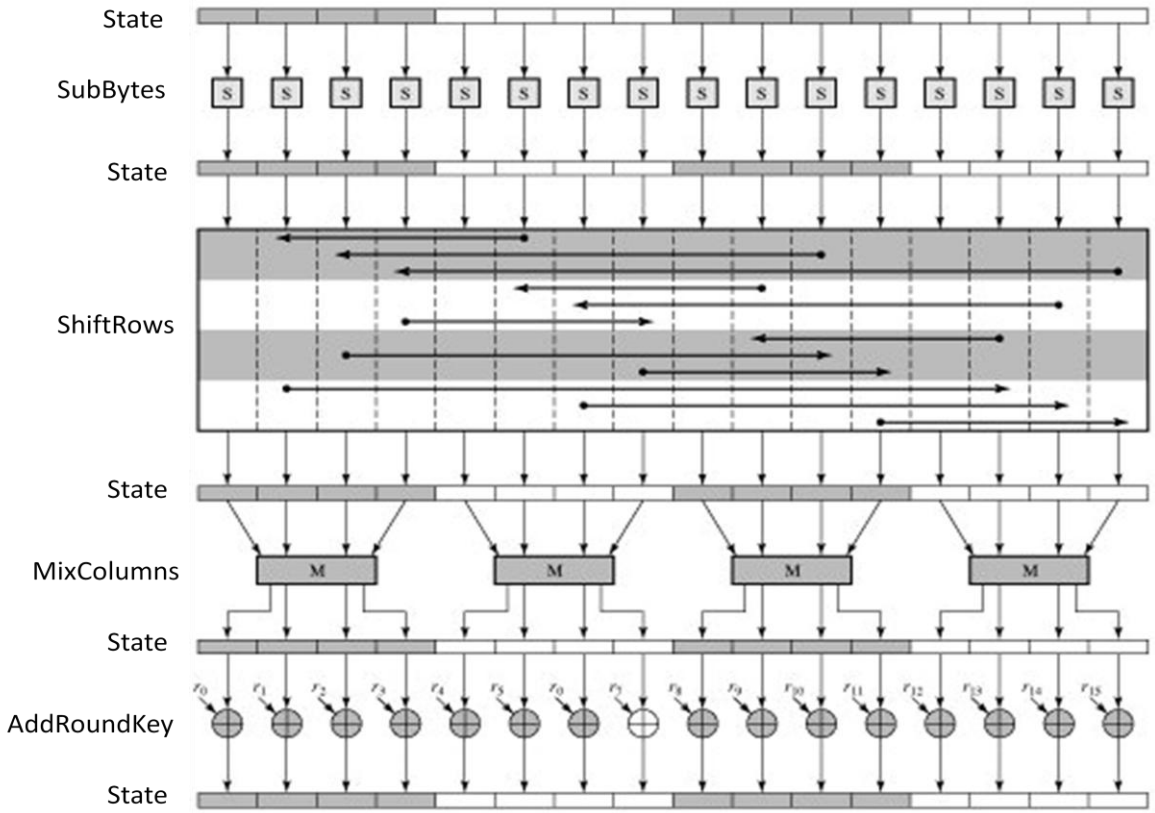


Figura 3.6 – Estrutura de um *round*de uma cifra completa no AES [21]

A seguir vai-se fazer referência a especificação do corpo finito *Galois Field* (GF 2^8) para uma melhor compreensão de todas as etapas usadas no AES.

Especificação *Galois Field* (GF 2^8)

No algoritmo AES, todos os bytes são interpretados como sendo elementos de um campo finito que podem ser representados por uma descrição polinomial, do tipo do que aparece na equação 3.1.

Equação 3.1

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

Por exemplo, o byte 57H com respetivo número binário 01010111, pode ser representado pelo polinómio presente na equação 3.2.

Equação 3.2

$$x^6 + x^4 + x^2 + x + 1$$

Desta forma torna-se possível toda uma descrição matemática de funções como adição, multiplicação, etc., tratando-se agora cada byte como um polinómio e efetuando tais operações polinomiais.

Na representação polinomial para somar bytes, devem-se somar os coeficientes dos termos com a mesma ordem de grandeza, desta forma o resultado da soma é dado por (bi +ai) mod 2, ou seja basta fazer um XOR entre ai e bi. No caso da multiplicação, esta é dada pelo resto da divisão do produto dos dois polinómios por um polinómio irredutível, ou seja, um polinómio de grau 8 é divisível por 1 e por ele próprio.

De seguida é feita uma discussãodo algoritmo de expansão de chaves, seguido da discussão de cada um dosquatro estágiosutilizado emAES, assim como a forma como foram implementados.Para cada etapa, é feita a descrição do algoritmode cifragem, e do algoritmo inverso (decifragem).

AES Key Expansion

Algoritmode Expansãoda chave

OalgoritmoAESde expansão da chave(*key expansion*) tem como entrada uma chave de 128 bits, que corresponde a 4palavras(16-bytes) de chave eproduz umamatriz linear de44 palavras(176bytes).
As 44 palavras são suficientes paraque seja possível fornecer 10 chaves de*round*de 4 palavras (128 bits) cada, uma para cada *round* da cifragem ou decifragem, onde são usadas nas transformações AddRoundKey.

O funcionamento geral do algoritmo de chave de expansão encontra-se representado na figura 3.7, onde se encontra ilustrada a geração das primeiras 8 palavras da chave expandida. Para se entender melhor o funcionamento deste algoritmo é importante referir que a chave é copiada para as primeiras 4 palavras da chave expandida, e a restante chave expandida é preenchida por 4 palavras de cada vez, onde cada palavra adicionada depende da palavra imediatamente anterior, e da palavra que se encontra 4 posições atrás. Onde em 3 dos casos é usado uma simples operação de XOR, mas no caso da palavra que ocupa a posição na matriz que é múltipla de 4, é usada uma função mais complexa, a função g.

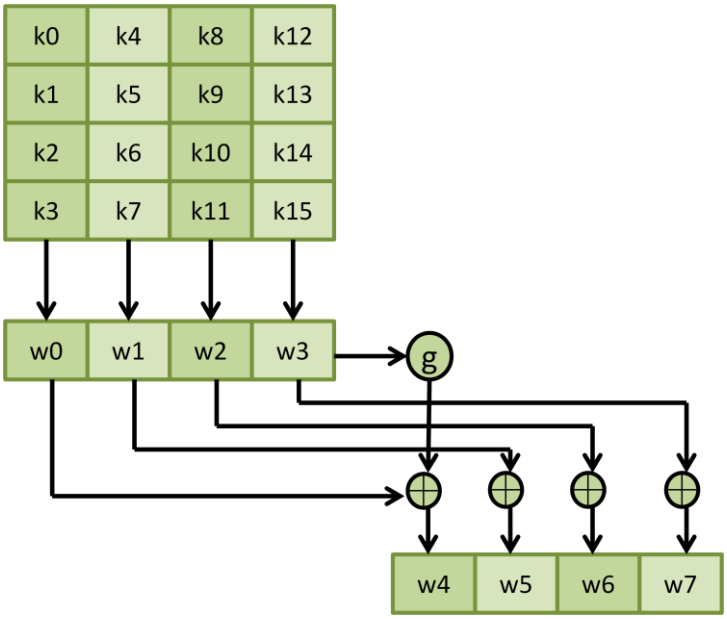


Figura 3.7 - Funcionamento geral da Key Expansion do AES

A função g é mostrada na figura 3.8 e consiste em 3 subfunções. A primeira subfunção é o RotWord, onde é feito um *shift* à esquerda numa palavra. A segunda subfunção é o SubWord que executa uma substituição de byte em cada byte da palavra de entrada, usando a Sbox, queconsiste numa matriz de 16 x 16 valores de byte, definida pelo AES, que contém

combinações de todos os 256 valores possíveis de 8 bits, tal como apresentado na Tabela 3.1.Por ultimo, com o resultados das subfunções anteriores é feito uma operação XOR com uma constante predefinida de *round*, chamada de Rcon[j].

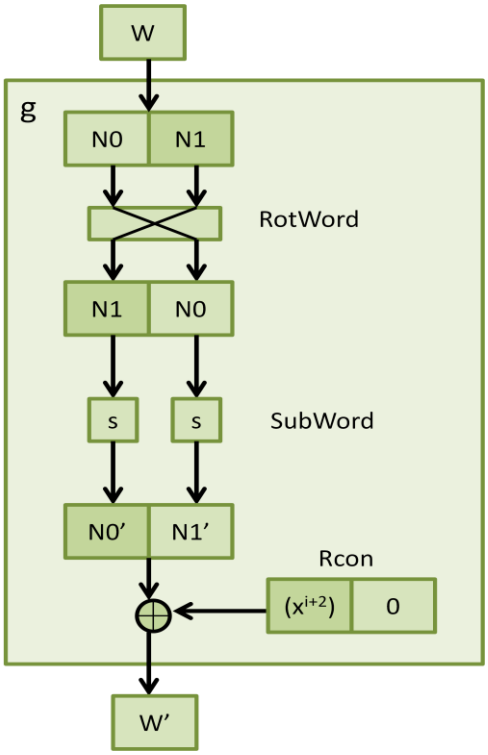


Figura 3.8 – Função g usado no Key Expansion.

É necessário entender a forma como a S-Box é constituída [21]. Inicialmente, a S-Box é inicializada com valores de byte linha por linha, em ordem ascendente, a primeira linha contém{00}, {01}, {02 },....{0F}; a segunda linha contém{10}, {11}, etc., e assim consecutivamente.Desta maneira, o valordo bytenalinha x, coluna yé {xy}, a seguir, é mapeado cada byte na S-Boxao seumultiplicador inverso nocampo finitoGF(2^8), o valor {00} é mapeado porsi mesmo.

Cada byte naS-box é composto por8 bitsclassificados, da seguinte forma, (b_7, b_6, b_5,b_4, b_3,b_2, b_1, b_0). Deve-se aplicar a seguinte transformação (Equação 5.3) em cada bit de cada bytenaS-Box:

Equação 3.3

$$b_i^{'} = b_i \oplus b_{(i+4) mod\ 8} \oplus b_{(i+5) mod\ 8} \oplus b_{(i+6) mod\ 8} \oplus b_{(i+7) mod\ 8} \oplus C_i$$

ondeciéobitda posiçãoith dobyteccom o valor{63}, isto é, $(c_7c_6c_5c_4c_3c_2c_1c_0) = (01100011)$.

Oprimo (')indica que a variáveldeve ser atualizadapelo valordo lado direito.

O AESretrataesta transformaçãona forma matricial,como se pode ver na equação 3.4.

Equação 3.4

$$\begin{pmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

A equação 3.4 deve ser interpretada com cuidado. Na multiplicação de matrizes comuns, cada elemento da matriz produto é a soma dos produtos dos elementos ou uma linha e uma coluna. Neste caso, cada elemento da matriz produto é um XOR bit a bit de produtos de elementos de uma linha e uma coluna. Além disso, a adição final mostrada na Equação 3.2 é um XOR bit a bit.

Tabela 3.1- AES Sbox

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9ª	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5ª	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6ª	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2ª	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3ª	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7ª	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8ª
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

O **inverso do substitute byte transformation**,chamado**InvSubBytes**, usa a S-Boxinversa mostrada naTabela3.2.

Tabela 3.2– AES Sbox inversa

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	52	09	6ª	D5	30	36	A5	38	FB	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1ª	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9ª	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7ª	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2ª	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

O inverso da S-box é construídoatravés da aplicação doinverso da transformaçãona Equação3.3, seguido pela adoção doinverso multiplicativoem GF(2^8).A transformaçãoinversaencontra-se presente na equação 3.5:

Equação 3.5

$$\boldsymbol{b_i^{'} = b_{(i+2)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+7)mod8} \oplus d_i}$$

Pode se confirmar que o InvSubBytes é o inverso do SubBytes olhando para o seguinte exemplo. Tem-se uma entrada {9A} que produz a saída{37} no InvSubBytes, a entrada{37} para aS-box produza saída {9A} no SubBytes.

A Rcon[j] é uma palavra onde os três bytes mais à esquerda são sempre 0, e byte mais a esquerda é o único que varia, sendo atribuído um diferente Rcon para cada *round*. A constante de *round*, pode ser vista da seguinte maneira $Rcon[j] = (RC[j], 0, 0, 0)$, com $RC[1] = 1, RC[j] = 2 \cdot RC[j-1]$e com a multiplicação definida sobre o campo GF(2^8).

Na tabela seguinte (3.3) encontram-se os valores de RC[j] em hexadecimal.

Tabela 3.3– Valor de RC[j] para cada *round.*

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

Nas figuras 3.9 e 3.10 estão presentes extratos de código Verilog do algoritmo de expansão de chave e da função g respetivamente.

```
/*As primeiras 4 palavras usam a chave de cifragem*/
        assign w[0] = {key[15],key[14],key[13],key[12]};
        assign w[1] = {key[11],key[10],key[9],key[8]};
        assign w[2] = {key[7],key[6],key[5],key[4]};
        assign w[3] = {key[3],key[2],key[1],key[0]};

/*As palavras seguintes usam a função g, e a operação XOR entre as chaves anteriores*/
        assign w[4] = g0_out ^ w[0];
        assign w[5] = w[4] ^ w[1];
        assign w[6] = w[5] ^ w[2];
        assign w[7] = w[6] ^ w[3];
(..)
/*São declarados as constantes Rcon para cada round, que são usadas na função g*/
        wire [31:0] rcon0 = {8'h01, 8'h00, 8'h00, 8'h00};
        g g0 (w[3], g0_out, rcon0);

        wire [31:0] rcon1 = {8'h02, 8'h00, 8'h00, 8'h00};
        g g1 (w[7], g1_out, rcon1);
(…)
```

Figura 3. 9–Extrato de código Verilog do Módulo KeyExpansion

```
/*É chamado o módulo Sbox (4 vezes) para mapear cada byte da palavra*/
Sbox sb1(S1, l1, c1);

wire [7:0]S2;
wire [3:0]l2, c2;

assign c2=R1[11:8];
assign l2=R1[15:12];
Sbox sb2(S2, l2, c2);
(..)
/*Os bytes de saida da Sbox são organizados de forma a que depois se efectue um XOR com a constante rcon*/
assign out_rw = {S4,S3,S2,S1};
assign out = out_rw ^ rcon;
(..)
```

Figura 3. 10 – Extrato de código Verilog do Módulo da função g

Substitute Bytes Transformation

A**Subtitute Byte Transformation**, chamada de **SubBytes**, é visto como uma tabela de pesquisa simples representada na Figura 3.11.

Qualquer byte pode ser substituído por um outro através da seguinte transformação: o *nibble* mais significativo representa uma linha na tabela e, o *nibble* menos significativo representa uma coluna, podendo assim indexar univocamente um valor da S-Box.

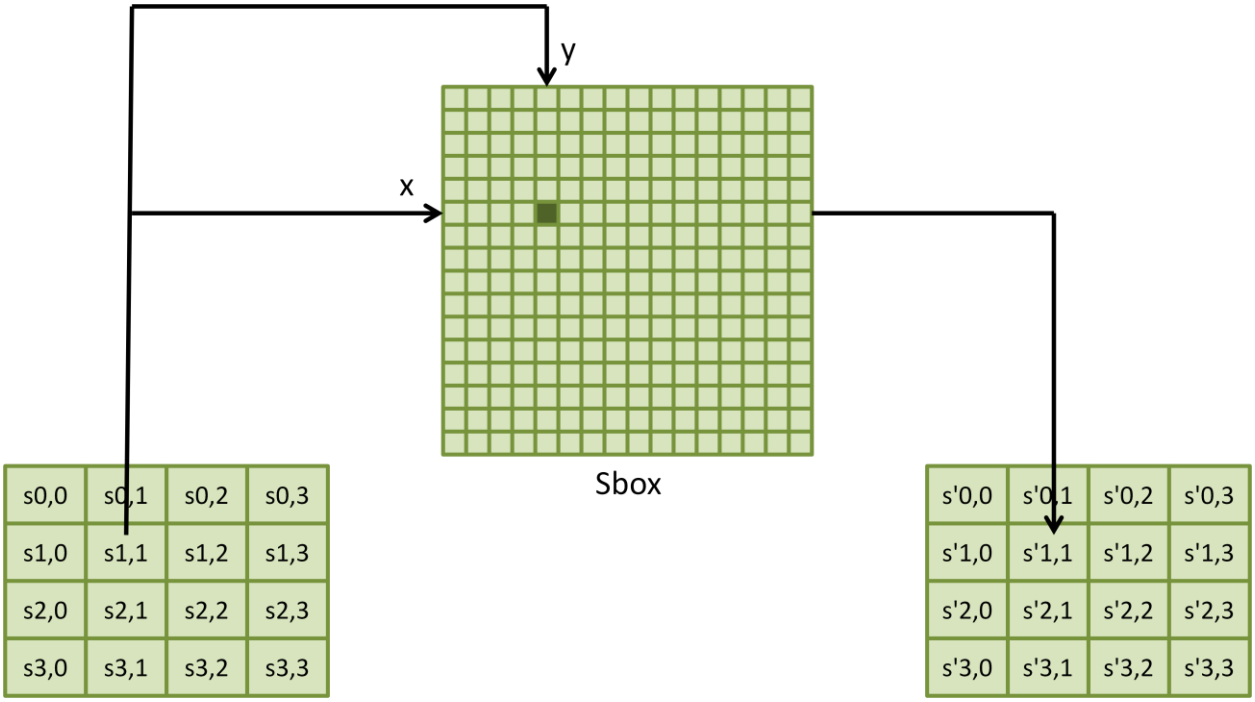


Figura 3. 11 – Funcionamento geral da etapa SuBytes

Por exemplo, o valor em hexadecimal {37} refere-se a linha 3, coluna 7 da S-Box, que contém o valor {9A}. Desta maneira, o valor {37} é mapeado no valor {9A}.

De seguida, será apresentado um exemplo mais concreto da transformação SubBytes (figura 3.12), onde a primeira matriz estão os 128 bits de dados de entrada da etapaSubBytes, e na segunda matriz o resultado da etapa SubByte:

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F9	2D	AD	C5

87	F2	4D	97
EC	6E	4C	90
4ª	C3	4	E7
8C	D8	95	A6

Figura 3. 12 – Exemplo da transformação SubBytes.

Na figura 3.13 é apresentado um extrato de código Verilog que foi elaborado para implementar a transformação SubBytes..

Pode-se verificar que os 128 bits da matriz estado, estão alocados numa matriz 4 x 4 valores de byte. Depois é mapeado cada byte num novo byte, para tal é chamado o módulo Sbox 16 vezes, uma para cada byte.

```
/*Os bits são ordenados corretamente numa matriz 4 x 4, com 8 bits em cada posição */
        wire [31:0]st [3:0];
        assign st[0] = {state[15], state[14], state[13], state[12]};
        assign st[1] = {state[11], state[10], state[9], state[8]};
        assign st[2] = {state[7], state[6], state[5], state[4]};
        assign st[3] = {state[3], state[2], state[1],state[0]};

wire [31:0] R1;
assign R1 = st[0];

wire [7:0]S1;
wire [3:0]l1, c1;

/*São atribuídos os primeiros 4 bits mais á esquerda do byte como valores de linha, e os 4 bits mais à direita como valores de coluna, estas operações são feitas 16 vezes*/
assign c1=R1[3:0];
assign l1=R1[7:4];

/*É chamado o módulo Sbox (16 vezes) para mapear cada byte da matriz*/
Sbox sb1(S1,l1,c1);
(…)
```

Figura 3.13–Extrato de código Verilog do Módulo SubBytes

ShiftRows Transformation

O **shift row transformation**, chamado de **ShiftRows**, consiste numa permutação simples, onde a primeira linhada matriz estadonão é alterada, na segunda linha é rodado um 1 byte para a

esquerda, naterceira linha são rodados 2 bytes para a esquerda, e por fim naquarta linha são rodados 3 bytes para a esquerda, como representado na figura seguinte (figura 3.14).

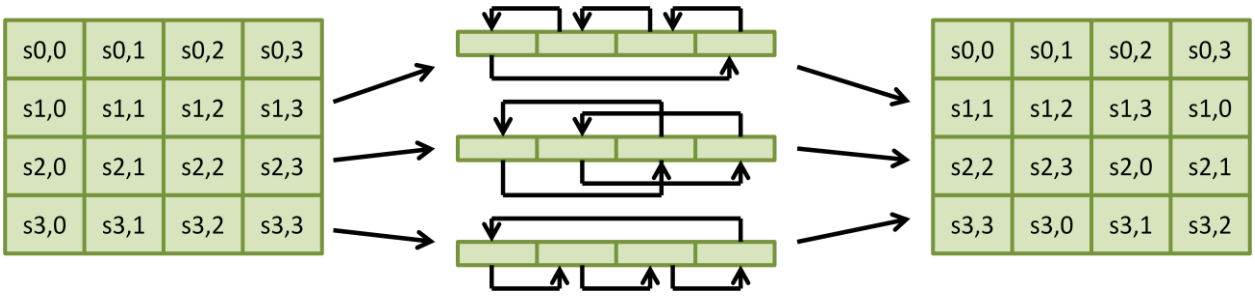


Figura 3.14 - Funcionamento geral da etapa ShiftRows

O simples funcionamento do ShiftRow pode ser visto no exemplo que se segue (figura 3.15):

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

87	F2	4D	97
6E	4C	90	EC
46	E7	4ª	C3
A6	8C	D8	95

Figura 3. 15 – Exemplo da transformação ShiftRow.

Na figura 3.16 é apresentado um extrato de código Verilog que foi elaborado para implementar a transformação ShiftRows, onde efetua as permutações assim referidas.

```
/*Primeira linha mantém se inalterada*/
        assign L1 = {state[15], state[14], state[13], state[12]};
/*Segunda linha é rodado um byte à esquerda*/
        assign L2 = {state[10], state[9], state[8], state[11]};
/*Terceira linha é rodado dois bytes à esquerda*/
        assign L3 = {state[5], state[4], state[7], state[6]};
/*Quarta linha é rodado três bytes à esquerda*/
        assign L4 = {state[0], state[3], state[2], state[1]};
(…)
```

Figura 3. 16 – Extrato de código Verilog do Módulo ShiftRow

O **inverso da shift row transformation**, chamado **InvShiftRows**, afetua os deslocamentoscircularesna direcção opostapara cada uma dastrês últimas linhas, rodando para a

direitaum bytena segunda linha, 2 bytes na terceira linha, e 3 bytes na quarta linha, ficando a primeira linha inalterada.

MixColumns Transformation

A **mix column transformation**, chamada de **MixColumns**, consiste numa substituiçãoque faz usoda aritméticasobreGF(2^8) e opera emcada colunaindividualmente.A transformação MixColumns pode ser definida pela matriz de multiplicação de estado, presente na figura 3.17, onde cada bytede uma colunaé mapeado num novo valor queé uma funçãode todos os quatrobytesnessa coluna.

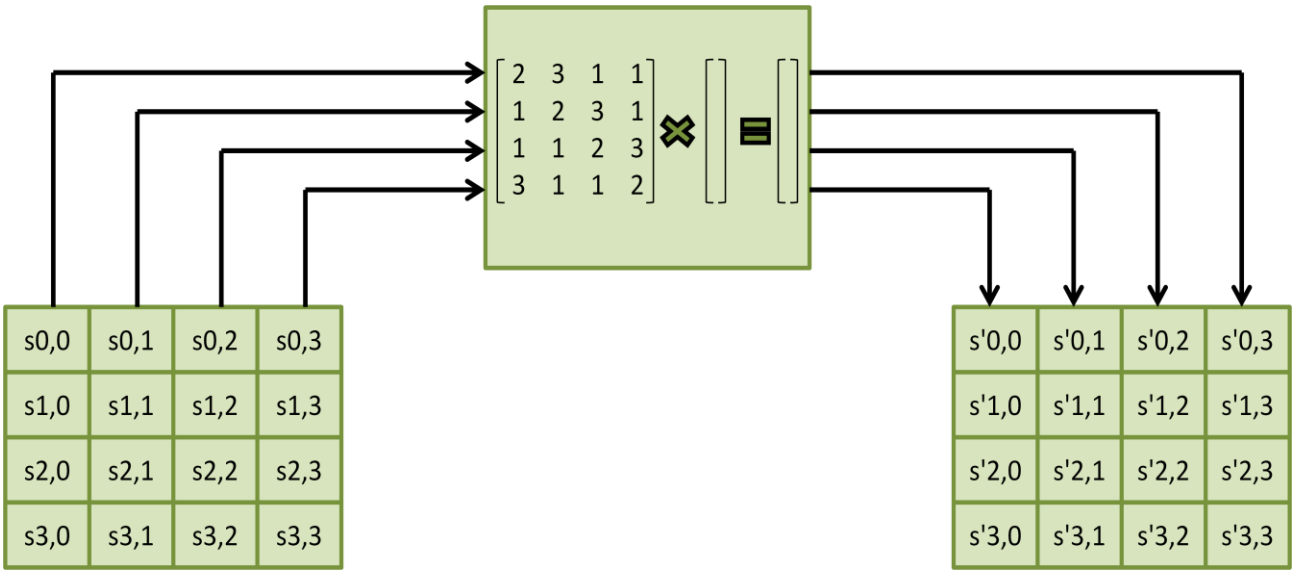


Figura 3. 17 -Funcionamento geral da etapa MixColumns

Na transformação MixColumns as adições individuais e multiplicações são realizadas em GF(2^8), como visto anteriormente. A transformada MixColumns é calculada usando a equação 3.6 a seguir:

Equação3.6

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Onde r_0, r_1, r_2 e r_3 é o resultado após a transformação, e a_0, a_1, a_2 e a_3 é a matriz que entrada da transformada que provém da etapa imediatamente anterior.

A equação 3.6 pode traduzir-se na equação 3.7 que deve ser bem analisada de forma a uma melhor compreensão dos processos seguintes da transformação MixColumns.

Equação3.7

$$\begin{bmatrix} \mathbf{02} & \mathbf{03} & \mathbf{01} & \mathbf{01} \\ \mathbf{01} & \mathbf{02} & \mathbf{03} & \mathbf{01} \\ \mathbf{01} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ \mathbf{03} & \mathbf{01} & \mathbf{01} & \mathbf{02} \end{bmatrix} \begin{bmatrix} \boldsymbol{s_{0,0}} & \boldsymbol{s_{0,1}} & \boldsymbol{s_{0,2}} & \boldsymbol{s_{0,3}} \\ \boldsymbol{s_{1.0}} & \boldsymbol{s_{1,1}} & \boldsymbol{s_{1,2}} & \boldsymbol{s_{1,3}} \\ \boldsymbol{s_{2,0}} & \boldsymbol{s_{2,1}} & \boldsymbol{s_{2,2}} & \boldsymbol{s_{2,3}} \\ \boldsymbol{s_{3,0}} & \boldsymbol{s_{3,1}} & \boldsymbol{s_{3,2}} & \boldsymbol{s_{3,3}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{s'_{0,0}} & \boldsymbol{s'_{0,1}} & \boldsymbol{s'_{0,2}} & \boldsymbol{s'_{0,3}} \\ \boldsymbol{s'_{1,0}} & \boldsymbol{s'_{1,1}} & \boldsymbol{s'_{1,2}} & \boldsymbol{s'_{1,3}} \\ \boldsymbol{s'_{2,0}} & \boldsymbol{s'_{2,1}} & \boldsymbol{s'_{2,2}} & \boldsymbol{s'_{2,3}} \\ \boldsymbol{s'_{3,0}} & \boldsymbol{s'_{3,1}} & \boldsymbol{s'_{3,2}} & \boldsymbol{s'_{3,3}} \end{bmatrix}$$

Usando a equação 3.7, pode ser expressar a matriz de estado em uma única colunaj ($0 \leq j \leq 3$), como apresentado nas equações (3.8) a seguir:

Equação 3.8

$$\boldsymbol{s'_{0,j}} = (\mathbf{2} \cdot \boldsymbol{s_{0,j}}) \oplus (\mathbf{3} \cdot \boldsymbol{s_{1,j}}) \oplus \boldsymbol{s_{2,j}} \oplus \boldsymbol{s_{3,j}}$$

$$\boldsymbol{s'_{1,j}} = \boldsymbol{s_{0,j}} \oplus (\mathbf{2} \cdot \boldsymbol{s_{1,j}}) \oplus (\mathbf{3} \cdot \boldsymbol{s_{2,j}}) \oplus \boldsymbol{s_{3,j}}$$

$$\boldsymbol{s'_{2,j}} = \boldsymbol{s_{0,j}} \oplus \boldsymbol{s_{1,j}} \oplus (\mathbf{2} \cdot \boldsymbol{s_{2,j}}) \oplus (\mathbf{3} \cdot \boldsymbol{s_{3,j}})$$

$$\boldsymbol{s'_{3,j}} = (\mathbf{3} \cdot \boldsymbol{s_{0,j}}) \oplus \boldsymbol{s_{1,j}} \oplus \boldsymbol{s_{2,j}} \oplus (\mathbf{2} \cdot \boldsymbol{s_{3,j}})$$

Em GF (2^8), a adição é uma operação XOR bit a bit, enquantoque a multiplicação existe uma regra estabelecida presente na equação (3.9) seguinte:

Equação 3.9

$$X * f(x) = \begin{cases} (b_6 b_5 b_4 b_3 b_2 b_1 b_0 0) & if\ b_7 = 0 \\ (b_6 b_5 b_4 b_3 b_2 b_1 b_0 0) \oplus (00011011) & if\ b_7 = 1 \end{cases}$$

Analisando a equação 3.9 é possível verificar que a multiplicação de um valor por x, pode ser implementado como um *shift* á esquerda de um bit seguido por uma condição de XOR bit a bit com (0001 1011), para o caso de o bit mais à esquerda do valor inicial (antes do *shift*) ser de 1, para o caso de o bit mais à esquerda do valor inicial ser de 0, não ocorre o *shift*.

Para uma melhor compreensão do funcionamento da transformação MixColumns segue-se o seguinte exemplo (figura 3.18):

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

Figura 3. 18 – Exemplo da transformação MixColumns.

Na primeira coluna da matriz à esquerda da figura 3.18 é preciso mostrar que seguindo a equação 3.8 se obtêm:

$$(\{02\} \cdot \{87\}) \oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\}$$
$$\{87\} \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\}$$
$$\{87\} \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\}$$
$$(\{03\} \cdot \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\}$$

Para a primeira equação têm-se$\{87\} =$ 1000 0111,$\{02\} =$ 0001 1011, $\{6E\} =$ 0110 1110, $\{46\} =$ 0100 0110e$\{A6\} =$ 1010 0110.

$\{02\} \cdot \{87\} =$ 1000 0111 $\ll$ 1(<< é o shift á esquerda, 1 é o numero do *shift* realizado)

$=$ 0000 1110 XOR 0001 1011 (porque o bit mais á esquerda é 1 antes do *shift*)

$=$ 0001 0101

$$\{03\} \cdot \{6E\} = \{10\ XOR\ 01\} \cdot \{0110\ 1110\}$$
$$= \{0110\ 1110 \cdot 10\}\ XOR\ \{0110\ 1110 \cdot 10\}$$

$= \{0110\ 1110 \cdot 10\}\ XOR\ \{0110\ 1110\}$ (Porque $\{0110\ 1110\} * 1$ [em decimal] = $\{0110\ 1110\}$)

$=$ 0110 1110 XOR 0001 1011 XOR 0110 1110 (não se faz o shift porque o bit mais á esquerda é 0, isto acaba por ser $\{6E\} \cdot \{02\}\ XOR\ \{6E\}$)

Desta forma, a equação fica:

$$\{02\} \cdot \{87\} = 0001\ 0101$$
$$\{03\} \cdot \{6E\} = 1011\ 0010$$
$$\{46\} = 0100\ 0110$$
$$\{A6\} = 1010\ 0110$$
$$0100\ 0111 = \{47\}$$

Na equação 5.7 o valor $s'_{0,0}$é substituído por {47}. Seguindo a mesma lógica demonstrada em cima é possível então obter:

$$\{87\} \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\} = \boldsymbol{s'_{0,1}}$$

$$\{87\} \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\} = \boldsymbol{s'_{0,2}}$$

$$(\{03\} \cdot \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\} = \boldsymbol{s'_{0,3}}$$

Asoutras colunasda matriz à esquerda da figura 3.18podemser da mesma formaverificadas.

Na figura 3.19, 3.20 e 3.21 são apresentadosextratos de código implementado correspondente a transformação MixColumns, onde aparecem todos os cálculos que foram referidos anteriormente.

```
/*São chamados 16 vezes o modulo m2 e m3, para ser possível efetuar as cálculos
necessários, que fazem uso da aritmética sobre GF(2^8)*/
m2 mm1 (state[15], tmp1);
m3 mn1 (state[11], tmp2);
m2 mm2 (state[11], tmp3);
m3 mn2 (state[7], tmp4);
m2 mm3 (state[7], tmp5);
m3 mn3 (state[3], tmp6);
m2 mm4 (state[3], tmp7);
m3 mn4 (state[15], tmp8);

/*Expressa-se a matriz de estado em uma única coluna conformo uma série de parâmetros,
esta operação é feita 4 vezes, uma para cada coluna (para as 16 posições)*/
        assign n1 = {tmp1 ^ tmp2 ^ state[7] ^ state[3]};
        assign n2 = {state[15] ^ tmp3 ^ tmp4 ^ state[3]};
        assign n3 = {state[15] ^ state[11] ^ tmp5 ^ tmp6 };
        assign n4 = { tmp8 ^ state[11] ^ state[7] ^ tmp7 };
(…)
```

Figura 3.19–Extrato de código Verilog do Módulo MixColumns

```
/*É realizada a operação de multiplicação por 2 em Galois Field (2^8)*/
        assign out = (in[7]) ? ((in << 1)^8'b00011011) : (in << 1);
```

Figura 3.20 - Extrato de código Verilog do Módulo m2 (multiplicação por 2 em GF)

```
/*Chama o modulo m2, que faz a multiplicação por 2 em GF*/
m2 m(in, tmp);

/*Multiplicação por 3 em GF, que consistem em multiplicar por 2 e depois fazer um XOR do valor da multiplicação com o byte de entrada*/
assign out = tmp ^ in;
```

Figura 3.21–Extrato de código Verilog do Módulo m3 (multiplicação por 3 em GF)

O **inverso mix column transformation**, chamado de **InvMixColumns**, é definido pelamultiplicação das matrizesseguintes:

Equação 3.10

$$\begin{bmatrix} \mathbf{0E} & \mathbf{0B} & \mathbf{0D} & \mathbf{09} \\ \mathbf{09} & \mathbf{0E} & \mathbf{0B} & \mathbf{0D} \\ \mathbf{0D} & \mathbf{09} & \mathbf{0E} & \mathbf{0B} \\ \mathbf{0B} & \mathbf{0D} & \mathbf{09} & \mathbf{0E} \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

Não é visível de imediato que a equação 3.10 é a inversa da equação 3.7, para tal é necessário verificar que a multiplicação entre a matriz de transformação e a matriz de transformação inversa é igual a matriz de identidade, como se pode ser na equação 3.11.

Equação 3.11

$$\begin{bmatrix} \mathbf{0E} & \mathbf{0B} & \mathbf{0D} & \mathbf{09} \\ \mathbf{09} & \mathbf{0E} & \mathbf{0B} & \mathbf{0D} \\ \mathbf{0D} & \mathbf{09} & \mathbf{0E} & \mathbf{0B} \\ \mathbf{0B} & \mathbf{0D} & \mathbf{09} & \mathbf{0E} \end{bmatrix} \begin{bmatrix} \mathbf{02} & \mathbf{03} & \mathbf{01} & \mathbf{01} \\ \mathbf{01} & \mathbf{02} & \mathbf{03} & \mathbf{01} \\ \mathbf{01} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ \mathbf{03} & \mathbf{01} & \mathbf{01} & \mathbf{02} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Usando a equação 3.11, pode ser expressar a matriz de estado em uma única coluna c,para $0 \leq c \leq 3$, como apresentado nas equações (3.12) a seguir:

Equação 3.12

$$s'(x) = a^{-1}(x) \oplus s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} \mathbf{0E} & \mathbf{0B} & \mathbf{0D} & \mathbf{09} \\ \mathbf{09} & \mathbf{0E} & \mathbf{0B} & \mathbf{0D} \\ \mathbf{0D} & \mathbf{09} & \mathbf{0E} & \mathbf{0B} \\ \mathbf{0B} & \mathbf{0D} & \mathbf{09} & \mathbf{0E} \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1.c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Os resultados da multiplicação, de uma coluna de 4 bytes são substituídos por:

$$s'_{0,c} = (\{0e\} \cdot s_{0,c}) \oplus (\{0b\} \cdot s_{1,c}) \oplus (\{0d\} \cdot s_{2,c}) \oplus (\{09\} \cdot s_{3,c})$$
$$s'_{1,c} = (\{09\} \cdot s_{0,c}) \oplus (\{0e\} \cdot s_{1,c}) \oplus (\{0b\} \cdot s_{2,c}) \oplus (\{0d\} \cdot s_{3,c})$$
$$s'_{2,c} = (\{0d\} \cdot s_{0,c}) \oplus (\{09\} \cdot s_{1,c}) \oplus (\{0e\} \cdot s_{2,c}) \oplus (\{0b\} \cdot s_{3,c})$$
$$s'_{3,c} = (\{0b\} \cdot s_{0,c}) \oplus (\{0d\} \cdot s_{1,c}) \oplus (\{09\} \cdot s_{2,c}) \oplus (\{0e\} \cdot s_{3,c})$$

AddRoundKey Transformation

No **add round key transformation**, chamado de **AddRoundKey**, consiste de um simples XOR bit a bit dobloco actualcom uma porçãoda chaveexpandida, isto é, os 128 bitsda matriz de estadosão usados em um XOR bit a bitcom os128 bitsda chave de *round*.Como se pode ver na figura 5.20, a transformação AddRoundKey realiza um XOR coluna por coluna entre a matriz de estado atual e a chave de round(wi) obtida pela função *key expansion.*

Figura 3. 22 - Funcionamento geral da etapa AddRoundKey

Segue-se um exemplo da transformação AddRoundKey, onde a primeira matriz é a de estado e a segunda é da de chave de *round* (figura 3.23).

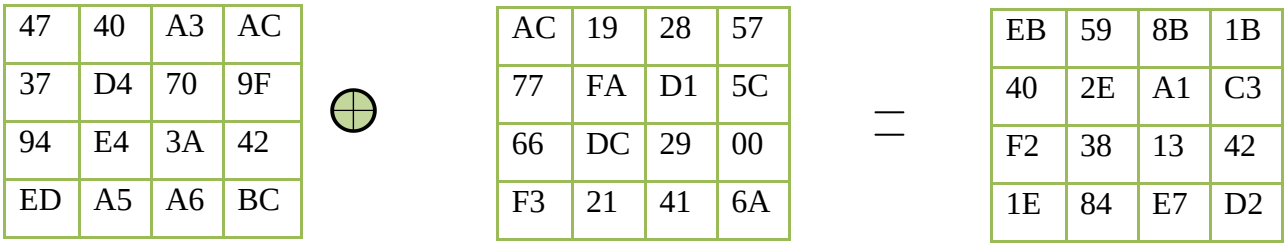


Figura 3. 23 – Exemplo da transformação AddRoundKey

Um extrato de código implementado da transformação AddRoundKey está presente na figura 3.24.

```
/*Operação XOR entre o bloco de dados e a chave de round*/
assign adr = plaintextfinal ^ keyfinal;
```

Figura 3. 24–Extrato de código Verilog do Módulo AddRoundKey

O **inversodo add round key transformation**chamado de invAddRoundKey é idêntico à transformação AddRoundKey,porque a operaçãoXORéo seu próprio inverso.

A segurança desta transformada reside na complexidade da chave expandida (key expansion).A transformação inversa do AddRoundKey étão simples quanto possívele afetatodos os bits da matriz de estado.A complexidadeda *key expansion round*, além da complexidade dasetapas daAES,garantem a segurança.

Decifragem (equivalente)

Como mencionado anteriormente, no algoritmo AES a decifragem não é idêntica à cifragem. As diferenças residem na sequência em que ocorrem as transformações dentro de cada *round*, as desvantagens que isso provoca é que são necessários dois módulos separados de *software* para aplicações que exigem encriptação e decifram de dados.
Nas figuras 3.25e 3.26 estão presentes extratos de código dos módulos de cifragem e decifragem respetivamente, onde aparece o primeiro *round* completo para se ver as diferenças.

```
/*Tem se como entrada o bloco de dados de 128 bits, a chave de cifragem de 128 bits e um clock, e como saída tem-se o bloco de dados cifrado*/
(…)
assign p1 = plaintext1;
assign kr1 = wout1 [1407:1280];

/*Inicialmente é feita a transformada AddRoundKey e gerada a chave expandida de 1408 bits, por forma a criar uma chave diferente por cada round*/
AddRoundKeyInicial ark1(p1,kr1,out1);

KeyExpansion ke(kin1,wout1);
(..)
/*Segue-se o primeiro round, onde passa pelas transformações SubBytes, ShiftRows, MixColumns e AddRoundKey*/
SubByte sb1(state1,out2);

ShiftRow sr1(state2,out3);

MixColumns mc1(state3,out4);
```

```
AddRoundKey ar1(state4,kr2,out5);

(…)
/*Seguiam-se agora os restantes 9 rounds, que não vão aparecer nesta imagem*/
```

Figura 3. 25– Extrato de código Verilog do Módulo AEScifragem

```
/*Tem se como entrada o bloco cifrado de 128 bits, a chave de decifragem de 128 bits e um clock, e como saída tem-se o bloco de dados original*/
(…)

assign p1 = ciphertext1;
assign kr1 = wout1 [1407:1280];
/*Inicialmente é feita a transformada invAddRoundKey e gerada a chave expandida de 1408 bits, por forma a criar uma chave diferente por cada round*/
invAddRoundKeyInicial ark1(p1, kr1, out1);

invKeyExpansion ke(kin1, wout1);

/*Segue-se o primeiro round, onde passa pelas transformações invShiftRows, invSubBytes, invAddRoundKey  e invMixColumns respectivamente*/
wire [127:0] out2;
invShiftRow isr1(state1, out2);

invSubByte isb1(state2, out3);

invAddRoundKey ar1(state3, kr2, out4);

invMixColumns imc1(state4, out5);

(…)
/*Seguiam-se agora os restantes 9 rounds*/
```

Figura 3. 26 – Extrato de código Verilog do Módulo AESdecifragem

3.2.1.1. AES com Triple Modular Redundancy

Os sistemas digitais são suscetíveis de falhas de execução como anteriormente referido, de forma a detetar e a tentar contrariar ou minimizar os efeitos dessas falhas foram implementadas técnicas de tolerância a falhas sobre o protocolo de encriptação AES. Neste caso, foi implementado *Triple Modular Redundancy* sobre o AES de forma a diminuir a vulnerabilidade dos sistemas a falhas transitórias causadas por diversos factos como colisão de partículas energéticas.

A Redundância Modular Tripla é a forma mais comum de redundância de *hardware*, conforme mencionado no capítulo 2. O conceito básico da TMR consiste em triplicar um módulo que se

deseja proteger e aplicar suas saídas a um *voter*. Esse *voter* reproduz a saída o valor lógico correspondente a pelo menos duas das suas entradas, conforme é possível verificar na figura 5.25. Desta forma, mesmo que um dos módulos apresente alguma falha, os resultados dos outros dois módulos irão mascarar a falha na saída do *voter*.

Como se pode verificar na figura 3.27, o *voter* consiste em três portas lógicas NAND e uma OR, de forma a que a votação seja feita pela maioria e resulte em uma única saída, desta forma se um dos sistemas falhar os outros dois iram mascarar a falha.

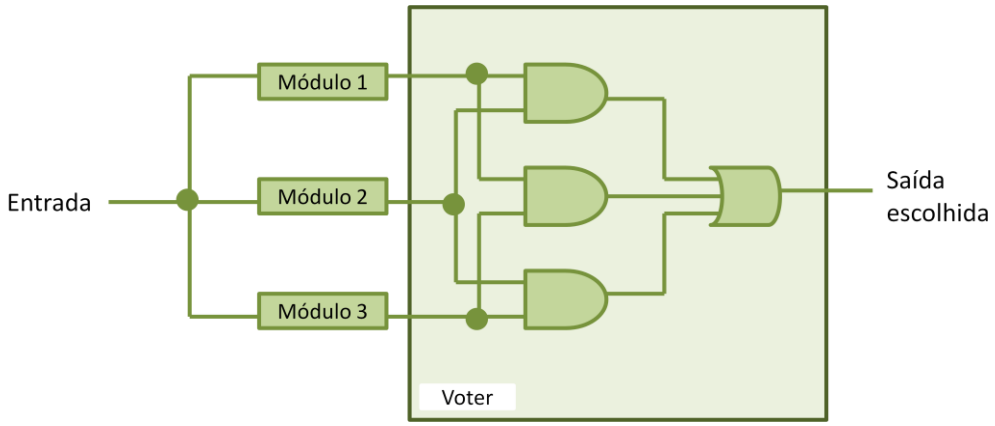


Figura 3. 27–Funcionamento do *Triple Modular Redundacy*

Na figura 3.28 está presente um extrato de código implementado do *voter*, onde é possível verificar a funcionalidade dele, anteriormente referida.

```
/*In1, In2 e In3 são os registos triplicados que realizam a mesma operação em paralelo*/
always@(In1 or In2 or In3)
begin
        for(i=0;i<WIDTH;i=i+1)
/*É feita um And lógico entre os três registos, seguido de um Or lógico onde o resultado é
colocado á saída*/
        Out[i] <= (In1[i] & In2[i]) | (In1[i] & In3[i]) | (In3[i] & In2[i]);
End
```

Figura 3. 28 – Extrato de código Verilog do Módulo *Voter*.

Em cada matriz de estado do protocolo de encriptação AES foram triplicadas as saídas e aplicado um *voter*de forma a que se ocorrer uma falha por ciclo está possa ser mascarada e o funcionamento do AES continue correcto. O funcionamento do AES (cifragem) com TMR encontra-se presente na figura 3.29.

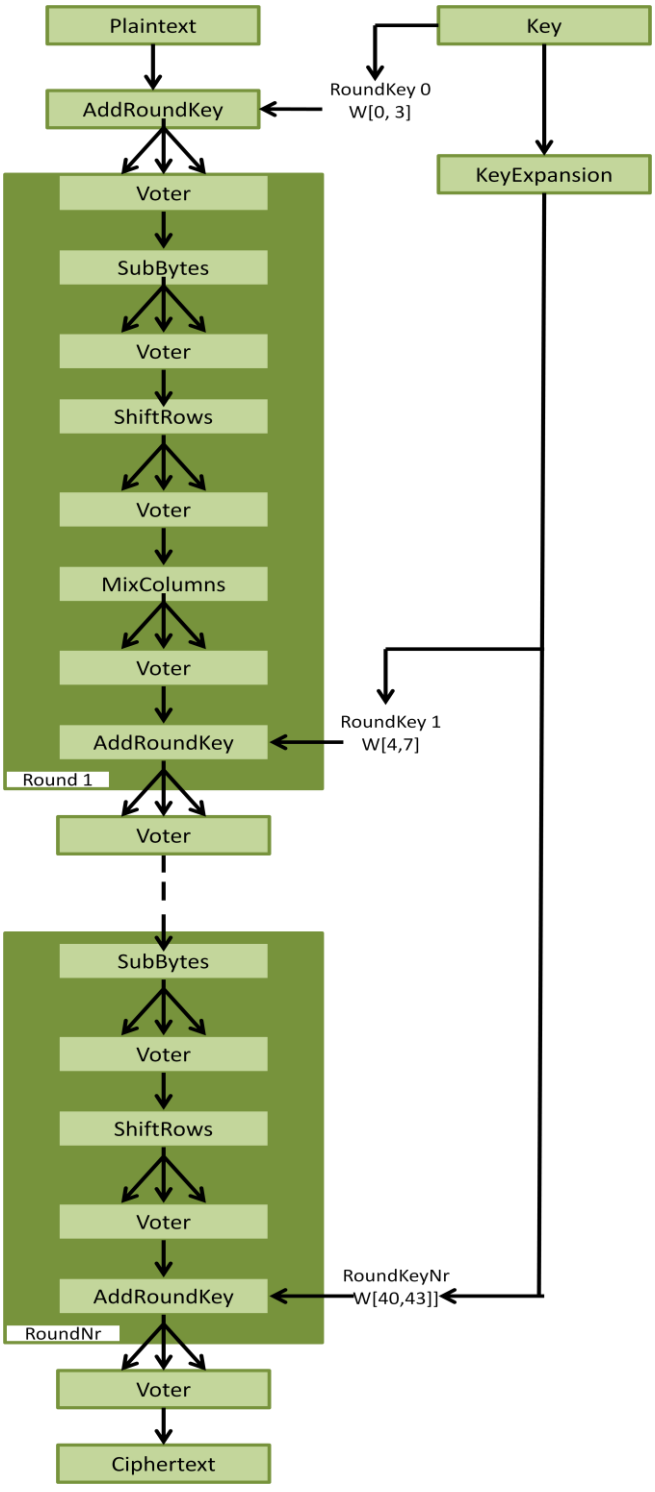


Figura 3.29 – Funcionamento do algoritmo de cifragem AES com TMR.

Como se pode ver, desta forma sempre que ocorrer uma falha por ciclo o correto funcionamento do algoritmo AES é assegurado, mas se a falha ocorrer no *voter* está não poderá ser mascarada, pois nessa parte do sistema não existem nenhum tipo de redundância, o que poderá resulta em uma falha em todo o sistema, logo o *voter* é um ponto único de falhas da técnica TMR, mas

como nesta dissertação o foco são as falhas SEUs, isto não produz um problema. No caso de ocorrerem 2 falhas num ciclo, pode levar a que dois módulos apresentem o mesmo erro, e neste caso o *voter* irá escolher o resultado incorreto levando a falha no algoritmo AES.

A grande desvantagem do TMR são os seus custos elevados, uma vez que implica *hardware* triplicado, mais votadores. Mais volume ocupado implicacustos económicos muito mais que triplicados.

3.2.1.2. AES com Códigos Hamming

Como alternativa ao TMR que usa redundância espacial, a redundância temporal parece bastante promissora já que em vez de triplicar o *hardware*, realiza a mesma operação várias vezes e compara os resultados.

No caso de estudo implementado foi decidido usar códigos de Hamming SEC-DED (*Single Error Correction- Double Error Detection*) sobre o algoritmo de encriptação AES, de seguida é feita uma descrição detalhada do funcionamento dos códigos Hamming e da forma como foi implementado.

Os códigos de Hamming são código bináriosdetetores e corretores de erros, e como tal devem satisfazer a seguinte equação:

Equação 3.13

$$d + p + 1 \leq 2^p$$

Na equação 3.13 o d corresponde ao número de bits de dados e o p corresponde ao número de bits de paridade. Desta forma, seguindo a equação é possível corrigir todos os erros de único bit em palavras de d bits e detetar erros de dois bits quando um bit de paridade total é usado.

A implementação do código Hamming é composta por dois blocos, por um bloco gerador que consiste em um bloco combinacional responsável por codificar os dados e pela inclusão de bits extra na palavra para a paridade, e por um bloco detetor que consistem em um bloco combinacional responsável por decodificar os dados, como se pode ver na figura 3.30.

Figura 3.30 – Funcionamento dos códigos Hammig.

O bloco gerador calcula os bits de paridade e pode ser implementado por um conjunto de portas lógicas XOR de duas entradas, que permitem a identificação de um único erro, como tal, são necessários seguir os seguintes passos para criar a palavra código. Inicialmente é necessário marcar todas as posições de bit que são potência de dois como bits de paridade (para este caso de estudo as posições são 1,2,4,8,16,32,64,128). As restantes posições de são para os dados que serão codificados, como por exemplo 3,5,6,7,9,etc. Cada bit de paridade calcula a paridade para alguns bits da palavra código. A posição do bit de paridade determina a sequência de bits que ele verifica ou ignora.

Por exemplo, para a posição 1, verifica 1 bit, ignora o bit seguinte, e assim consecutivamente, desta forma os bits verificados são:

1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,…. 123,125.127,129,131,133,135.

Para o caso da posição 2, verifica 2 bits e ignora os 2 bits seguintes, desta forma:

2,3,6,7,10,11,14,15, 18,19,22,23, 26,27,…. 122,123,126,127,130,131,134,135.

Para a posição 4, verifica 4 bits e ignora os 4 bits seguintes, da seguinte maneira;

4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31,…. 124,125,126,127,132,133,134,135.

Para o caso da posição 8, verifica 8 bits e ignora os 8 bits seguintes, desta forma:

8-15, 24-31, 40-47, 56-63, 72-79, 88-95, 104-111, 120-127, 136.

Para a posição 16, verifica 16 bits e ignora os 16 bits seguintes, da seguinte maneira:

16-31, 48-63, 80-95, 112-127.

Para a posição 32, verifica 32 bits e ignora os 32 bits seguintes, da seguinte maneira:

32-63, 96-127.

No caso da posição 64, verifica 64 bits e ignora os 64 bits seguintes, como se pode verificar a seguir:

64-127.

Para a posição 128, verifica 128 bits, e ignora os 128 bits seguintes, para este caso de estudo ficaria:

128-136.

Por último para a posição 137 é feito uma operação XOR dos bits 1 até 136.

O bloco detetor é mais complexo que o bloco gerador, porque além de ter que detetar a falha, também precisa de a corrigir. É composto pela mesma lógica que calcula a paridade mais um descodificador que indica o endereço do bit que contém o erro. Desta maneira, o bloco detetor é composto por um conjunto de portas XOR de duas entradas e algumas portas AND e inversores. Se todos os bits de paridade estiverem a 0 a palavra está correta. Se pelo menos um dos bits de

paridades for 1, existe uma inversão de bit. A posição do bit invertido é calculada pela concatenação dos bits de paridade P9P8P7P6P5P4P3P2P1 e lê-lo como um número binário original. Uma forma simples de entender isto é usar um exemplo mais simples de 4 bits de dados, com 3 bits de paridade, onde se deseja o número de código 6 (0110), a sua codificação é de 1100110. Introduzindo uma inversão de bit obtém-se 1100010. Os bits de paridade são calculados onde P1 = 1, P2 = 0 e P3 = 1, ao ler-se a concatenação de P3P2P1 tem se 101, desta forma é possível verificar que o erro encontra-se na posição 101.

Cada registo protegido de cada etapa do AES tem a entrada ligada ao bloco gerador e a saída ligada ao bloco detetor, uma vez que foi implementado um código Hamming SEC-DED, foram adicionados 9 bits de paridade uma vez que tem se blocos de dados de 128 bits, desta forma sempre que ocorrer uma falha por ciclo no algoritmo AES com códigos Hamming, essa falha é corrigida, mas se ocorrerem duas falhas por ciclos estas não são corrigidas mas são detetadas dando a posição exata onde ocorreram essas falhas.
O funcionamento do AES (cifragem) com códigos Hamming encontra-se presente na figura 3.31.

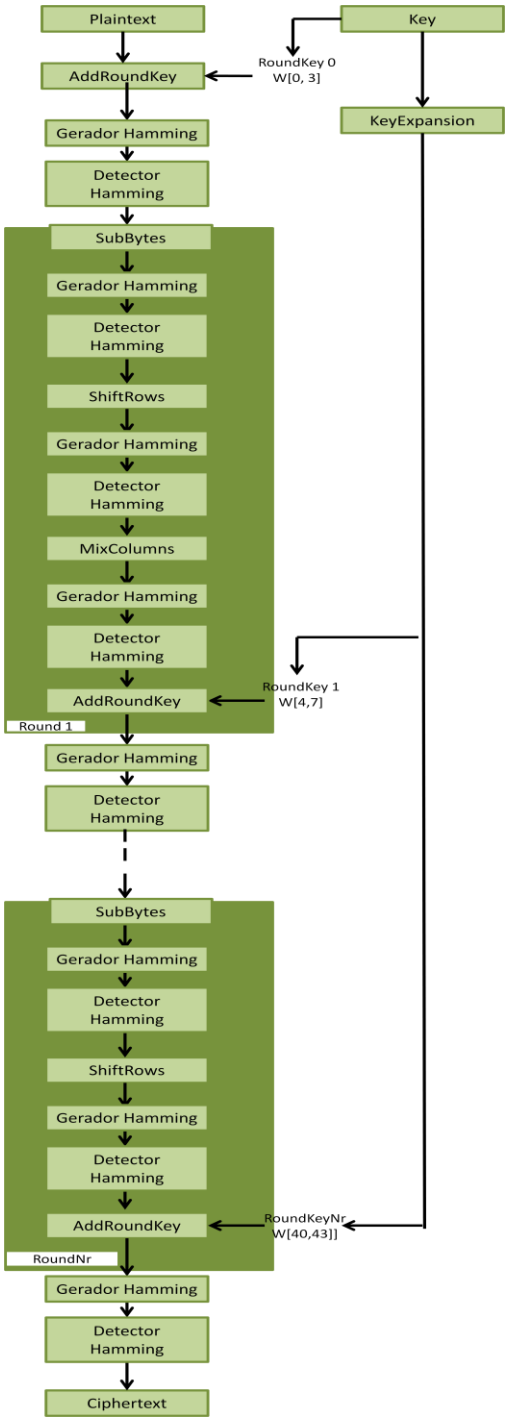


Figura 3.31 – Funcionamento do algoritmo de cifragem AES com códigos Hamming.

3.2.2. AES extensão

Ao AES implementado em Verilog foi adicionada tolerância a falhas como aconteceu anteriormente, mas desta vez usando as novas metodologias implementadas, isto é, no código AES entre cada etapa foram usados "reg_tmr" de forma a ser possível usar *Triple Modular Redundancy*, o código torna-se assim de implementação mais simples e legível, tornando o

código muito mais compacto, como é possível verificar na figura 3.32, que está representado um extrato de código do primeiro *round* completo do módulo de cifragem do AES.

```
/*Inicialmente é feita a transformada AddRoundKey e gerada a chave expandida de 1408 bits, por forma a criar uma chave diferente por cada round*/
AddRoundKeyInicial ark1(p1, kr1, out1);

KeyExpansion ke(kin1, wout1);
(..)

/*O reg_tmr  triplica os registos e depois usa um votador para fazer uma votação dos registosr*/
reg_tmr [127:0] vt1_out;
always@(posedge Clk)
begin
        vt1_out <= out1;
end
/*Segue-se o primeiro round, onde passa pelas transformações SubBytes, ShiftRows, MixColumns e AddRoundKey*/
SubByte sb1(state1, out2);
reg_tmr [127:0] vt2_out;
always@(posedge Clk)
begin
        vt2_out <= out2;
end

ShiftRow sr1(state2, out3);
reg_tmr [127:0] vt3_out;
always@(posedge Clk)
begin
        vt3_out <= out3;
end

MixColumns mc1(state3, out4);
reg_tmr [127:0] vt4_out;
always@(posedge Clk)
begin
        vt4_out <= out4;
end

AddRoundKey ar1(state4, kr2, out5);
(…)
/*Seguiam-se agora os restantes 9 rounds, que não vão aparecer nesta imagem*/
```

Figura 3.32 – Extrato de código Verilog do Módulo AEScifragem com extensão (ficheiro de entrada do pré processador de registos TMR)

O código apresentado na figura 3.32 não é sintetizável, uma vez que o reg_tmr não é traduzível em *hardware*, visto que ele foi desenvolvido para simplificar a codificação do *Triple Modular*

Redundancy, desta forma o reg_tmr é usado num o ficheiro de entradapara o SystemVerilog/Verilog pré processador de registos TMR, após esse ficheiro ser passado pelo pré processador o ficheiro de saída obtido é usado pelo simulador.

Na figura 3.33 encontra-se representado um extrato de código Verilog do ficheiro de saída do pré processador de registos TMR.

```
AddRoundKeyInicial ark1(p1,kr1,out1);

KeyExpansion ke(kin1,wout1);

reg  [127:0] vt1_out1, vt1_out2, vt1_out3;
wire [127:0] vt1_out;
voter #(128) voter_vt1_out(vt1_out,vt1_out1,vt1_out2,vt1_out3);

always@(posedge Clk)
begin
vt1_out1 <= out1;
vt1_out2 <= out1;
vt1_out3 <= out1;
end
wire [127:0] out2;

SubByte sb1(vt1_out,out2);
reg  [127:0] vt2_out1, vt2_out2, vt2_out3;
wire [127:0] vt2_out;
voter #(128) voter_vt2_out(vt2_out,vt2_out1,vt2_out2,vt2_out3);

always@(posedge Clk)
begin
vt2_out1 <= out2;
vt2_out2 <= out2;
vt2_out3 <= out2;
end
wire [127:0] out3;

ShiftRow sr1(vt2_out,out3);
(…)
```

Figura 3.33 – Extrato de código Verilog do Módulo AEScifragem com extensão.

4. Resultados e discussão

Este capítulo descreve os resultados obtidos após a implementação do caso de estudo e das novas metodologias de extensão à linguagem e a respetiva discussão.

Após desenvolvido o protocolo de encriptação AES foram obtidos os seguintes resultados de simulação usando o simulador ISIM da Xilinx as figuras 4.1 e 4.2 demonstram o resultado da cifragem e decifragem de dados.

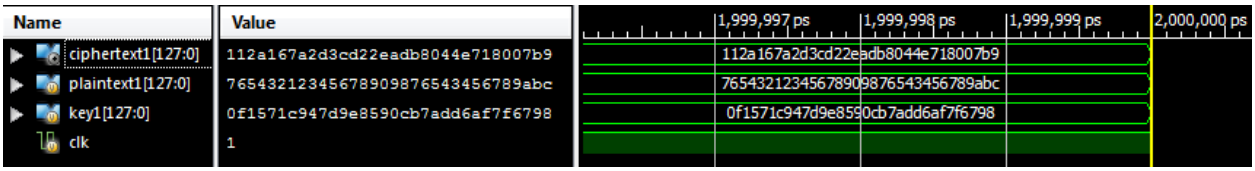


Figura 4.1 - Simulação do algoritmo AES cifragem.

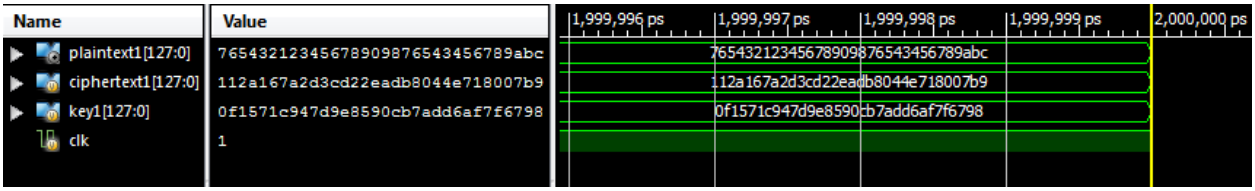


Figura 4.2 - Simulação do algoritmo AES decifragem.

Após a síntese do algoritmo AES cifragem para FPGA Xilinx Virtex 5 é possível verificar na tabela 4.1a informação sobre a utilização do sistema, onde se pode ver o número de registos usados, o número de LUTs, entre outras características, assim como a frequência máxima do sistema.

Tabela 4.1– Informação da utilização do AES.

AES simples	
Registers	4950
Total LUTs	16223
LUT FF-pairs	3925
I0s	385
Frequency	295.229MHz

Foram realizados testes ao AES inserindo 1, 2, 4, 8 até 512 falhas por ciclo (10 resultados de simulação por cada número de falhas injetadas por ciclo) de forma a verificar o número de erros propagados pelo sistema. Os resultados obtidos encontram-se presentes nas tabelas 4.2 e 4.3; a tabela 4.3 apresenta os resultados em forma percentual.

Tabela 4. 2– Resultados do número de erros propagados em função do número de falhas injetadas para o AES simples.

n.º de falhas injetadas por ciclo	n.º de falhas injetadas	n.º de erros propagados
1	49960	31632.4
2	99920	43272
4	199840	49037.6
8	399680	48837.7
16	799360	49957.6
32	1598720	49958.2
64	3197440	49958.9
128	6394880	49959
256	12789760	49959
512	25579520	49959

Tabela 4. 3– Resultados do número de erros propagados em função do número de falhas injetadas em percentagem para o AES simples.

n.º de falhas injetadas por ciclo	% n.º de falhas injetadas	% n.º de erros propagados
1	49960	63.31545236
2	99920	43.30664532
4	199840	24.53843074
8	399680	12.21920036
16	799360	6.24969976
32	1598720	3.12488741
64	3197440	1.562465597
128	6394880	0.781234362
256	12789760	0.390617181

512	25579520	0.195308591

A seguir estão apresentados os gráficos (figura 4.3 e 4.4) que representam os resultados obtidos das tabelas 4.2 e 4.3 respetivamente. Como se pode observar na figura 4.3 quando o número de falhas propagadas ronda os 199840 (4 falhas por ciclo) o número de erros propagados estabiliza porque foi atingido o ponto de saturação.

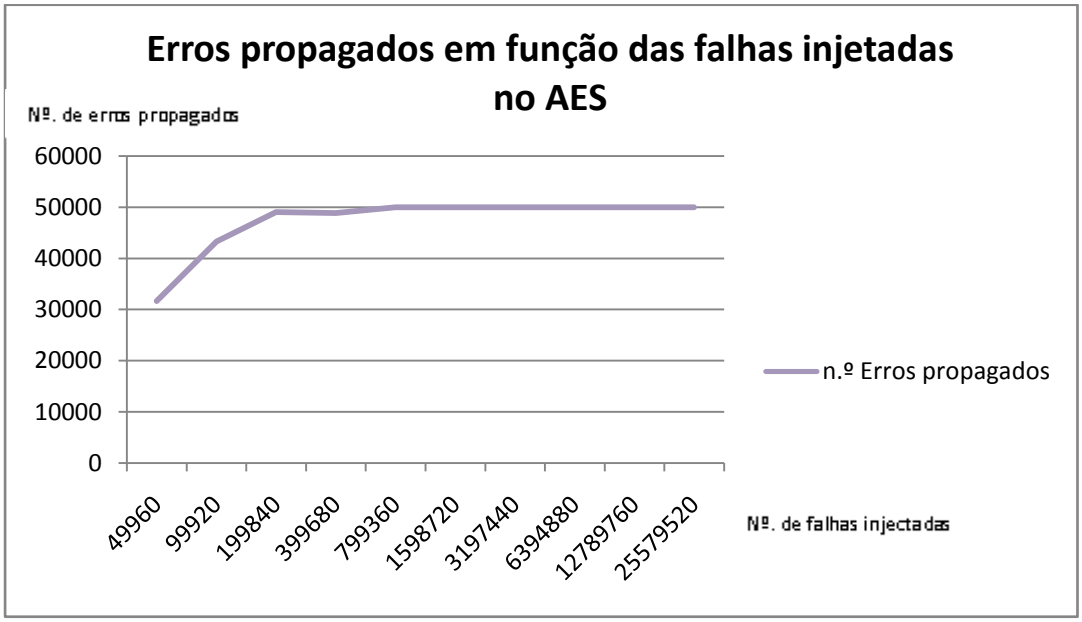


Figura 4. 3 –Gráfico linearque representa o número de erros propagados em função ao número de falhas injetadas.

Figura 4.4 – Gráfico linearque representa o número de erros propagados em função ao número de falhas injetadas.

Para o caso do AES com TMR os resultados de simulação usando o simulador ISIM da Xilinx encontram-se presentes nas figuras 4.5 e 4.6 onde é possível observar o resultado da cifragem quando nenhuma falha é injetada, e com 1 falha injetada por ciclo. Como se pode confirmar, quando é inserida uma falha por ciclo esta não produz erro nenhum.

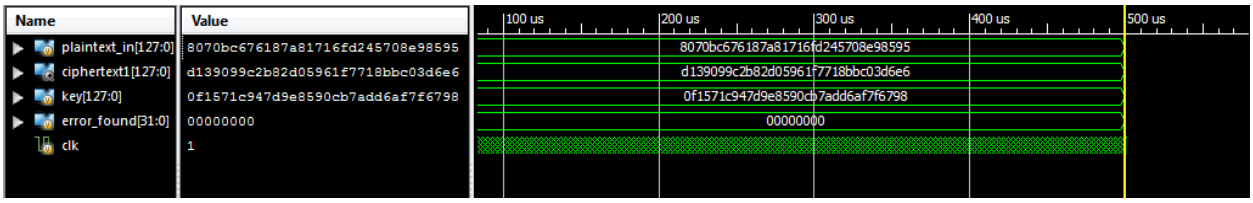


Figura 4.5 - Simulação do algoritmo AES cifragem com TMR sem injeção de falhas.

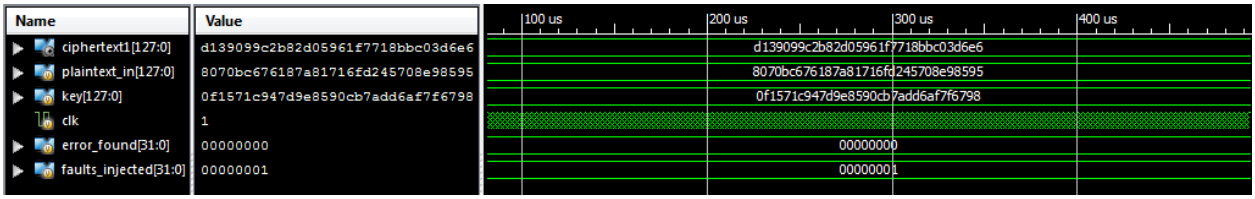


Figura 4.6 - Simulação do algoritmo AES cifragem com TMR com injeção de uma falha por ciclo.

Após a síntese do algoritmo AES cifragem com TMR para FPGA Xilinx Virtex 5 é possível verificar na tabela 4.4 a informação sobre a utilização do sistema, onde se pode ver o número de registos usados, o número de LUTs, entre outras características, assim como a frequência máxima do sistema. É importante reparar que o número de registos do AES com TMR em relação ao AES sem tolerância a falhas triplicou.

Tabela 4.4– Informação da utilização do AES (cifragem) com TMR.

AES cifragem com TMR	
Registers	15360
Total LUTs	27303
LUT FF-pairs	7190
IOs	385
Frequency	291.008MHz

Tal como aconteceu no caso do AES simples, para o AES com TMR também foram feitos testes inserindo 1, 2, 4, 8 até 512 falhas por ciclo (10 resultados de simulação por cada numero de falhas injetadas por ciclo) de forma a verificar o número de erros propagados pelo sistema, os resultados obtidos encontram-se presentes nas tabelas4.5 e 4.6, onde na tabela 4.6os resultados são apresentados em forma percentual.

Tabela 4.5– Resultados do número de erros propagados em função do número de falhas injetadas para o AES com TMR.

n.º de falhas injetadas por ciclo	n.º de falhas injetadas	n.º de erros propagados
1	49960	0
2	99920	7.4
4	199840	37
8	399680	177.8
16	799360	765
32	1598720	3086.1
64	3197440	11400.2
128	6394880	32107.5
256	12789760	49081.6
512	25579520	49956.5

Tabela 4.6– Resultados do número de erros propagados em função do número de falhas injetadas em percentagem para o AES com TMR.

n.º de falhas injetadas por ciclo	% n.º de falhas injetadas	% n.º de erros propagados
1	49960	0
2	99920	0.007405925
4	199840	0.018514812
8	399680	0.044485588
16	799360	0.095701561

32	1598720	0.193035679
64	3197440	0.356541483
128	6394880	0.502081353
256	12789760	0.383757006
512	25579520	0.195298817

A seguir estão apresentados os gráficos (figura 4.7 e 4.8) que representam os resultados obtidos na tabela 4.5 e 4.6 respetivamente. Como se pode observar na figura 4.8 quando o número de falhas propagadas ronda os 6394880 (128 falhas por ciclo) o número de erros propagados começa a estabilizar isto porque foi atingido um ponto de saturação, onde a partir deste momento ocorre o caso de que os bits que já estavam se encontram em estado de erro podem ser alterados e ficar num estado não erróneo.

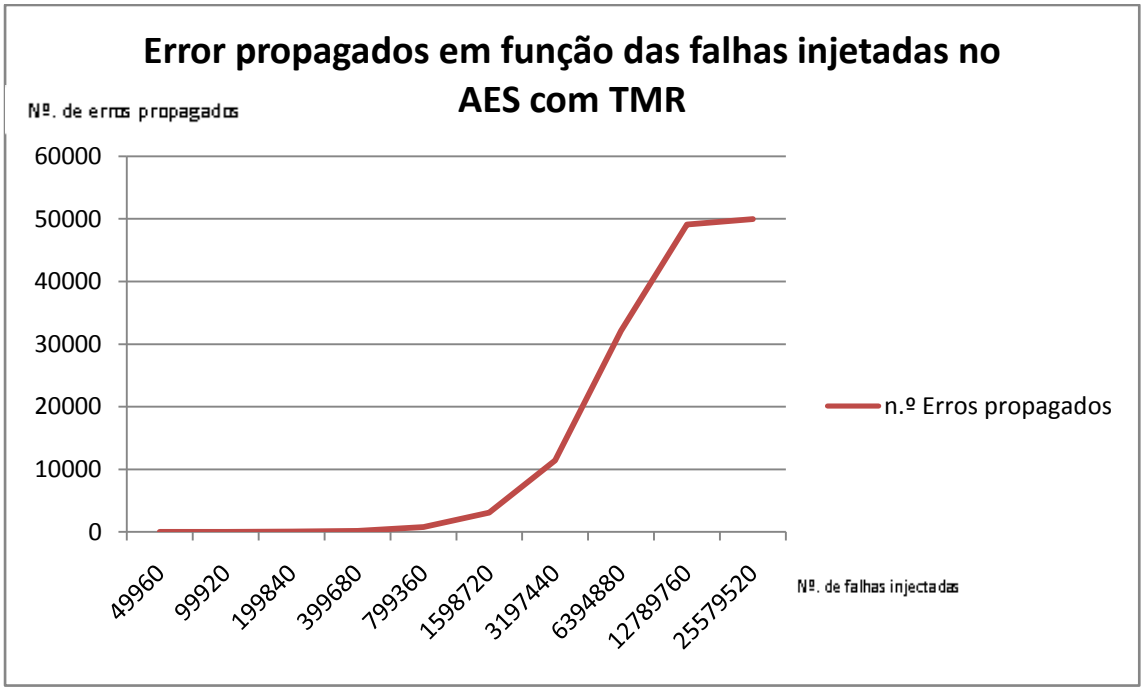


Figura 4.7 – Gráfico linear que representa o número de erros propagados em função ao número de falhas injetadas para o AES com TMR.

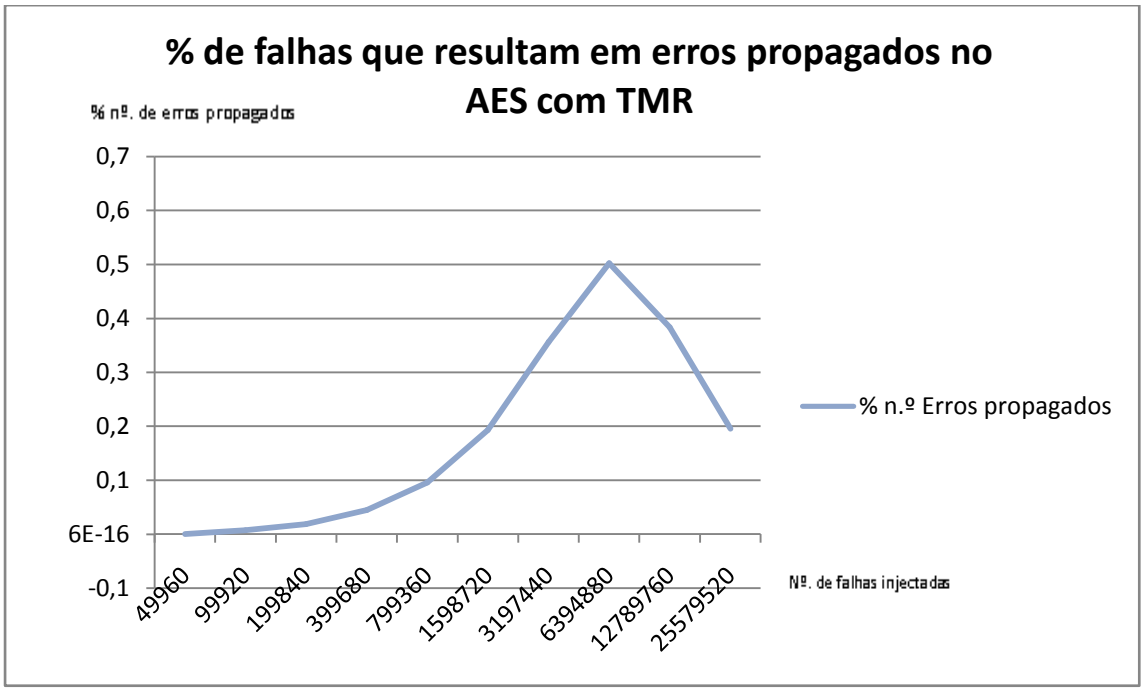


Figura 4. 8 – Gráfico linear que representa o número de erros propagados em função ao número de falhas injetadas para o AES com TMR.

É possível verificar que usando AES com TMR existem grandes custos a nível de área ocupada, devido ao número de registos ser triplicado.

Para o caso do AES com códigos Hamming (SECDED) os resultados de simulação usando o simulador ISIM da Xilinx encontram-se presentes nas figuras 4.9, 4.10 e 4.11 onde é possível observar o resultado da cifragem quando não existe nenhuma falha, com uma falha e com duas falhas respetivamente. Como se pode verificar, quando ocorre uma falhaestaé corrigida, mas quando ocorrem 2 falhas estas são detetadas mas não corrigidas.

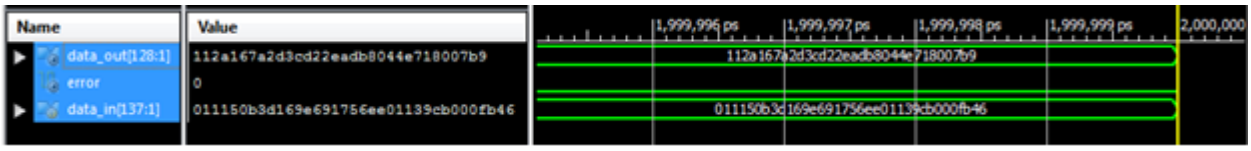


Figura 4. 9 - Simulação do algoritmo AES cifragem com códigos Hamming sem erros.

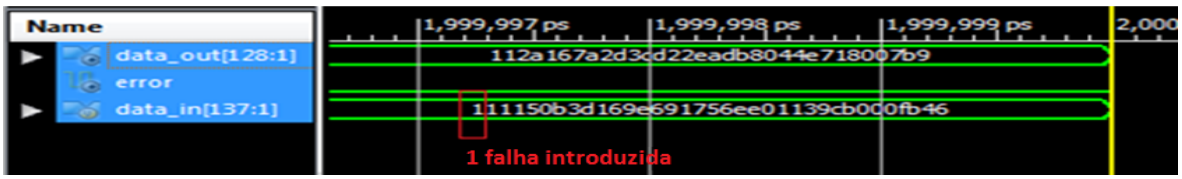


Figura 4. 10 - Simulação do algoritmo AES cifragem com códigos Hamming com um erro introduzido.

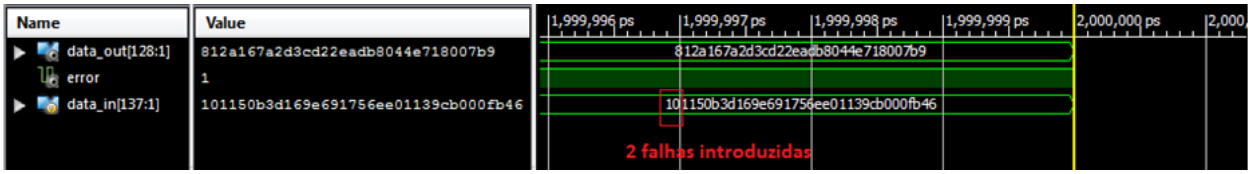


Figura 4. 11 - Simulação do algoritmo AES cifragem com códigos Hamming com dois erros introduzidos.

Após a síntese do algoritmo AES cifragem com Hamming para FPGA Xilinx Virtex 5 é possível verificar na tabela 4.4 a informação sobre a utilização do sistema, onde se pode ver o número de registos usados, o número de LUTs, entre outras características, assim como a frequência máxima do sistema. É importante reparar que o número de registos do AES com TMR em relação ao AES sem tolerância a falhas triplicou.

Tabela 4.7– Informação da utilização do AES (cifragem) com códigos Hamming**.**

AES cifragem com códigos Hamming	
Registers	4384
Total LUTs	24191
LUT FF-pairs	2784
IOs	425
Frequency	108.528MHz

Tal como aconteceu no caso do AES simples e no AES com TMR, para o AES com códigos Hamming também foram feitos testes inserindo 1, 2, 4,8 até 512 falhas por ciclo (10 resultados de simulação por cada numero de falhas injetadas por ciclo) de forma a verificar o número de erros propagados e detetados pelo sistema, os resultados obtidos encontram-se presentes nas tabelas 4.8 e 4.9, onde na tabela 4.8 é apresentado os resultados em percentagem, e na tabela 4.8 aparecem ainda o numero de erros total que consiste na soma dos erros propagados com os erros detetados.

Tabela 4.8– Resultados do número de erros propagados, detetados e totais em função do número de falhas injetadas para o AES com códigos Hamming.

n.º de falhas injetadas por ciclo	n.º de falhas injetadas	n.º de erros propagados	nº. de erros detetados	nº. erros TOTAL
1	49960	0	0	0
2	99920	0	1217.3	1217.3
4	199840	101.4	6941.9	7043.3
8	399680	1103.3	29137.5	30240.8
16	799360	4740.6	102887.8	107628.4
32	1598720	9940.2	294158.2	302098.4
64	3197440	16277.1	621130	637407.1
128	6394880	22173.1	893747	915020.1
256	12789760	24856	965821	990677
512	25579520	25109.1	970011.1	995120.2

Tabela 4.9– Resultados do número de erros propagados e detetados em função do número de falhas injetadas, valores obtidos em percentagem para o AES com códigos Hamming.

n.º de falhas injetadas por ciclo	% n.º de falhas injetadas	% n.º de erros propagados
1	49960	0
2	99920	0.007405925
4	199840	0.018514812
8	399680	0.044485588
16	799360	0.095701561
32	1598720	0.193035679
64	3197440	0.356541483
128	6394880	0.502081353
256	12789760	0.383757006
512	25579520	0.195298817

A seguir estão apresentados os gráficos que representam os resultados obtidos na tabela 4.5 e 4.6. Na figura 4.12 e 4.13 aparecem representados o número de erros propagados em função do número de falhas injetadas, onde na figura 4.13 os valores vêm em percentagem, e nela é possível observar que quando o número de falhas propagadas ronda os 3197440 (64 falhas por ciclo) o número de erros propagados começa a baixar, isto porque foi atingido um ponto de saturação, onde a partir deste momento ocorre o caso de que os bits que já estavam se encontram em estado de erro podem ser alterados e ficar num estado não erróneo.

Figura 4. 12 – Gráfico linear que representa o número de erros propagados em função ao número de falhas injetadas para o AES com códigos Hamming.

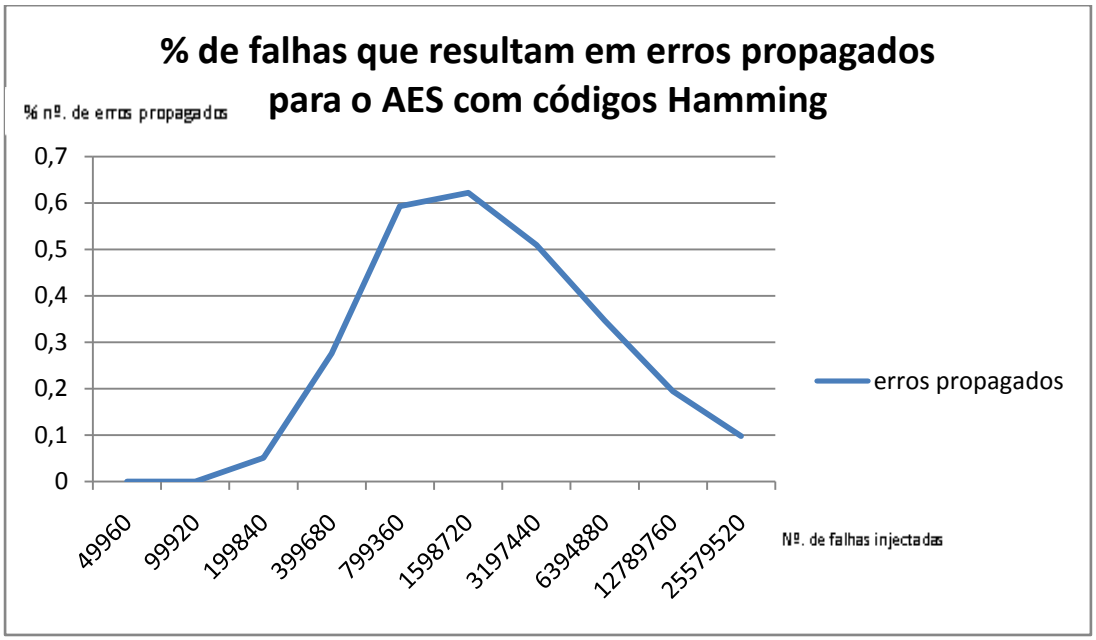


Figura 4.13 – Gráfico linear que representa o número de erros propagados em função ao número de falhas injetadas para o AES com códigos Hamming.

As figuras 4.14 e 4.15 que se seguem representam o numero de erros propagados em função do número de falhas injetadas para o AES com códigos Hamming. A figura 4.16 por sua vez representa o número total de erros, que consiste na soma dos erros detetados com os erros propagados.

Figura 4.14 – Gráfico linear que representa o número de erros detetados em função ao número de falhas injetadas para o AES com códigos Hamming.

Figura 4.15 – Gráfico linear que representa o número de erros detetados em função ao número de falhas injetadas para o AES com códigos Hamming.

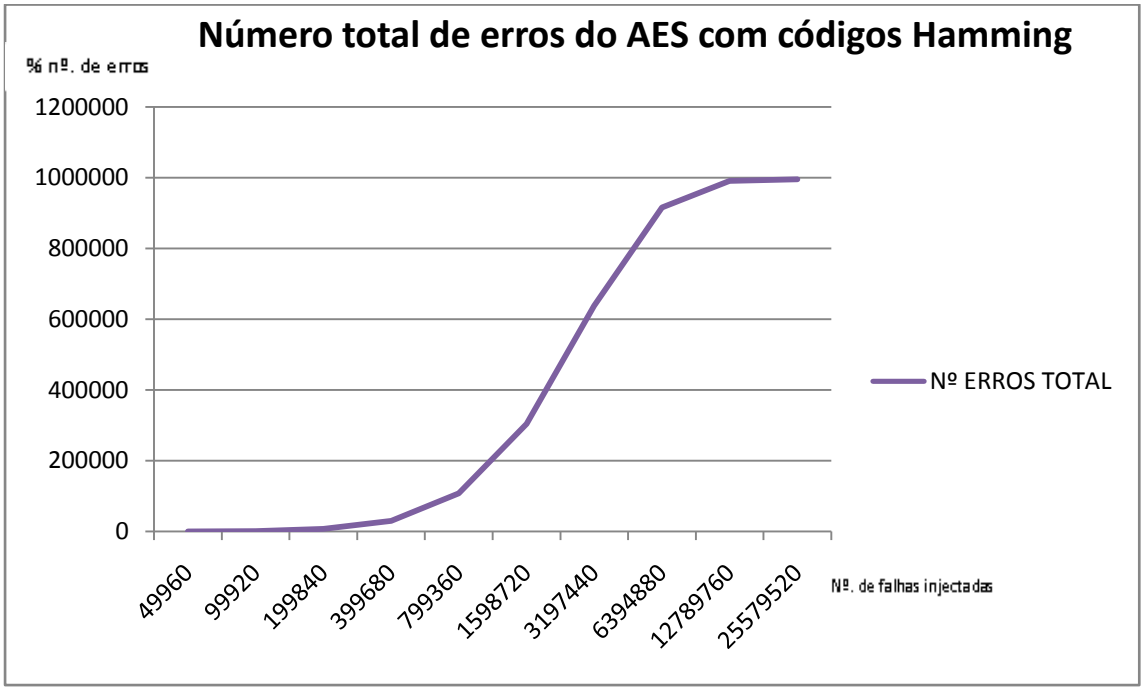


Figura 4. 16 – Gráfico linear que representa o número de erros total em função ao número de falhas injetadas para o AES com códigos Hamming.

Como é possível verificar, o AES com Hamming Codes tem um menor custo a nível de área, mas traz custos elevados a nível de performance, visto que as operações demoram N vezes mais tempo a serem realizadas.

A solução ideal seria um compromisso entre os custos de área e de performance.

Foi desenvolvido um demonstrador do algoritmo de encriptação AES, implementado em uma FPGA Xilinx Virtex 5, que é programada através de comunicação serie com o computador usando o protocolo rs232, como é possível verificar na figura4.17.

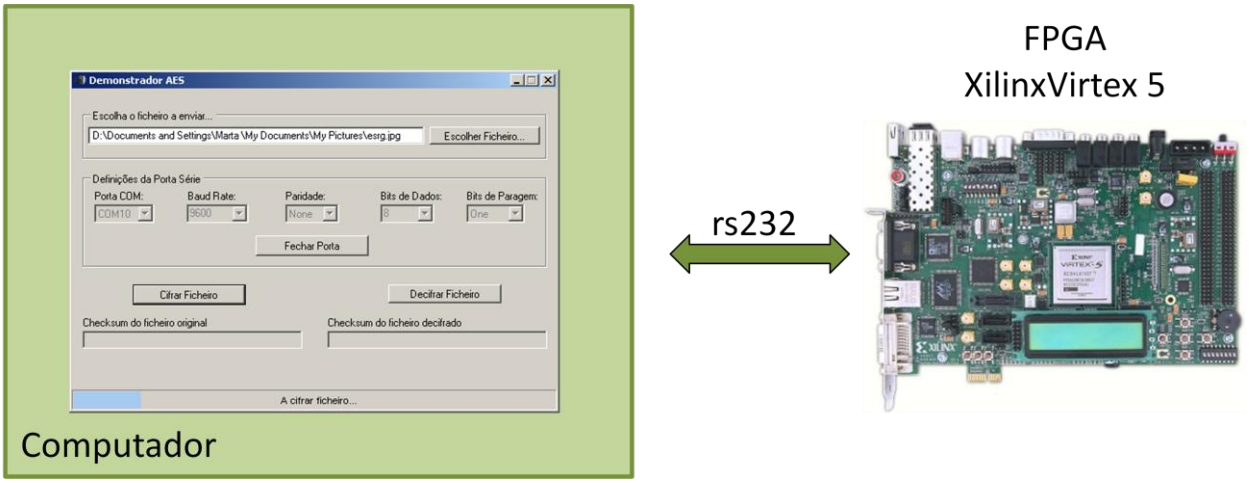


Figura 4.17 – Demonstrador AES.

O demonstrador consiste em uma UART (umrecetor (Rx) e um transmissor (Tx), 1 start bit, trama de 8 bits, 1 stop bit e *baudrate* de 9600 bits/s) e de uma máquina de estados (figura 4.18) que tem 3 estados principais (a receber, a encriptar e a transmitir). 0estado“a receber” recebe 8 bits de cada vez do pino Rx e guarda-os num *buffer* de 128 bits. Quando o buffer estiver cheio passa ao estado “a encriptar”, onde tanto pode ser feita a cifragem como a decifragem, selecionado a partir de um seletor na FPGA Xilinx Virtex 5, desta forma o utilizador pode escolher à sua vontade. Após os 128 bits serem cifrados ou decifrados passa para o estado “a transmitir”, onde são enviados de 8 em 8 bits através do pino Tx.

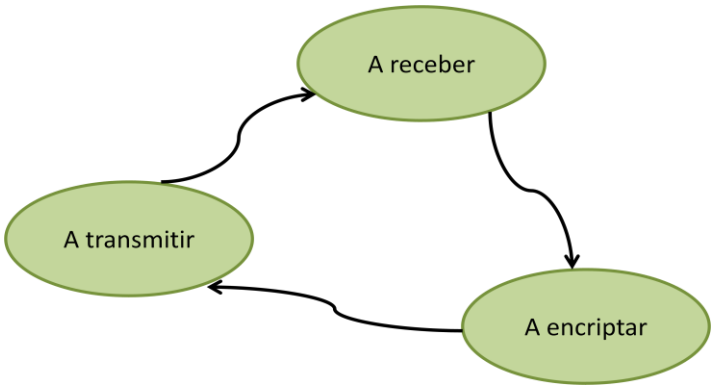


Figura 4.18 – Máquina de estados usada no demonstrador.

Na figura 4.19 está presente um extrato de código Verilog do demonstrador que representa a máquina de estados.

```
always@(posedge Clk)
begin
	if(!Rst)
		state <= `RECEIVING;
	else
	begin
		case(state)
		`RECEIVING:
		begin
			if(input_buffer_full)
			state <= `CRYPTING;
		end
		`CRYPTING:
		begin
			if(crypting_complete)
				state <= `TRANSMITTING;
		end
		`TRANSMITTING:
		begin
			if(Transmit_complete)
				state <= `RECEIVING;
		end
		endcase
	end
end
```

Figura 4.19 – Extrato de código Verilog do Módulo Demonstrador (máquina de estados).

Para a comunicação com o computador usa-se um programa em C que foi desenvolvido de forma a ser possível enviar todo o tipo de ficheiros de qualquer tamanho, uma vez que é necessário para a cifragem e decifragem que os ficheiros sejam múltiplos de 16 bytes, o programa encarrega-se de acrescentar bits de *padding*, cujo número é guardado numa variável interna ao programa. Após o ficheiro ser cifrado ele vai ter um tamanho diferente do inicial, sendo os bits de *padding*

retirado do ficheiro após a decifragem. Para certificar que o ficheiro decifrado é igual ao ficheiro original é feito o *checksum* que é apenas o MD5.

De seguida será feita uma breve apresentação do funcionamento do demonstrador. Aa FPGA Xilinx Virtex 5 já se encontra programada, e o *switch* seletor está a 0 isto é vai ser feita uma cifragem de uma imagem que tem por nome ersg.jpg (figura 4.20) que é a seguinte:

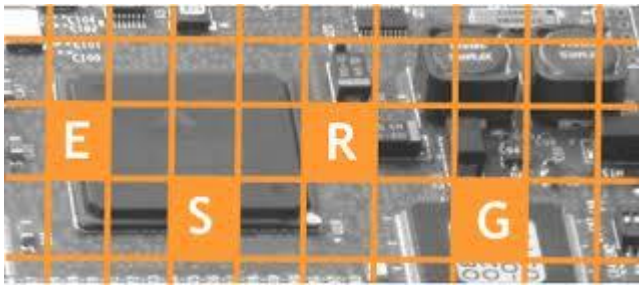


Figura 4.20 – Ficheiro original.

De seguida é selecionado então o ficheiro desejado, metem-se as definições desejadas da porta séria, e seleciona-se o botão cifrar ficheiro, como é possível ver na figura 4.21.

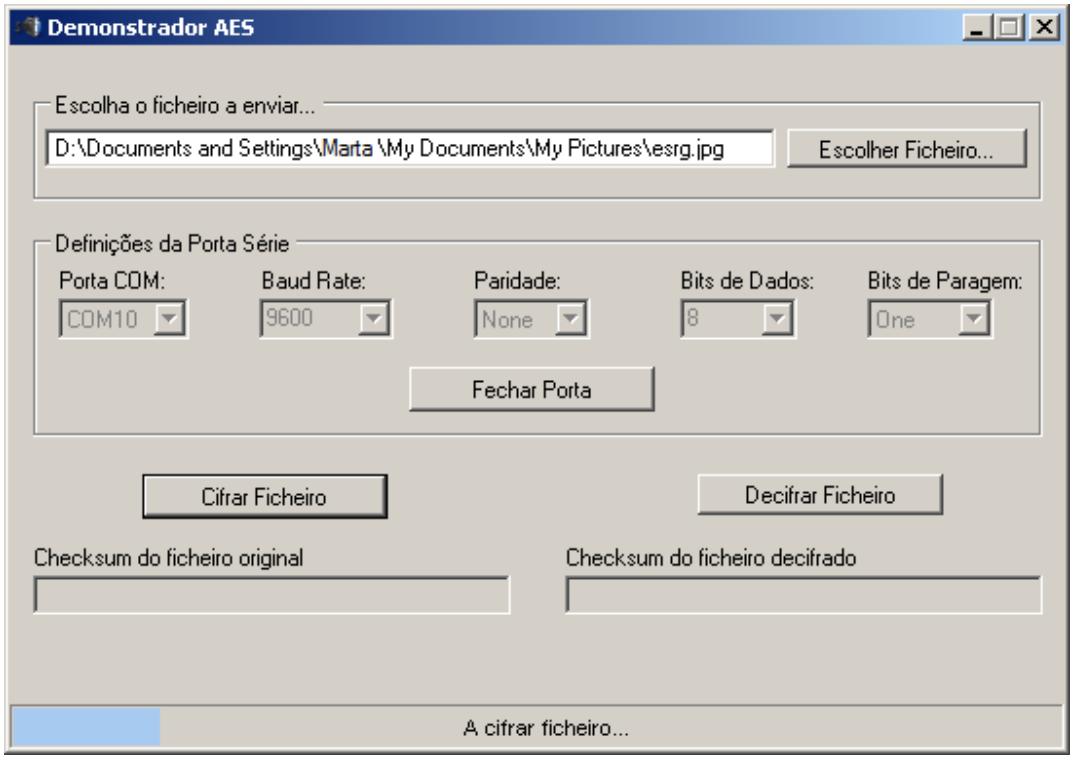


Figura 4.21 – Demonstrador AES a cifrar.

Após a cifragem concluir é gerado um *checksum* do ficheiro original, e é obtido um ficheiro de saída ilegível como se vê na figura 4.22.

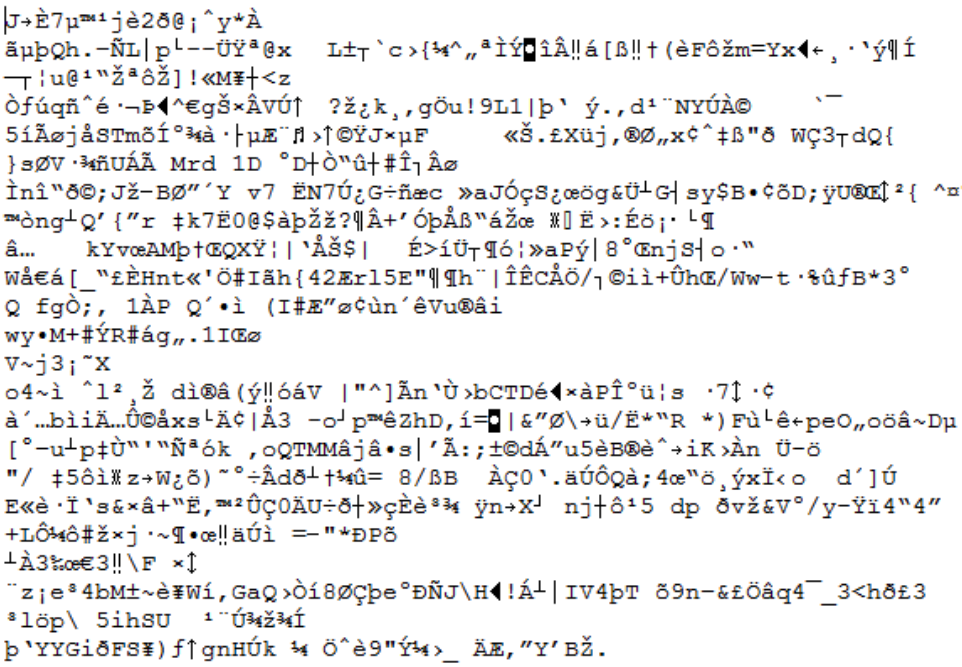


Figura 4.22 – Ficheiro cifrado.

Para a decifragem o processo é o mesmo, o *switch*deve ser selecionado a 1, e é selecionado o botão decifrar ficheiro, como é possível ver na figura 4.23.

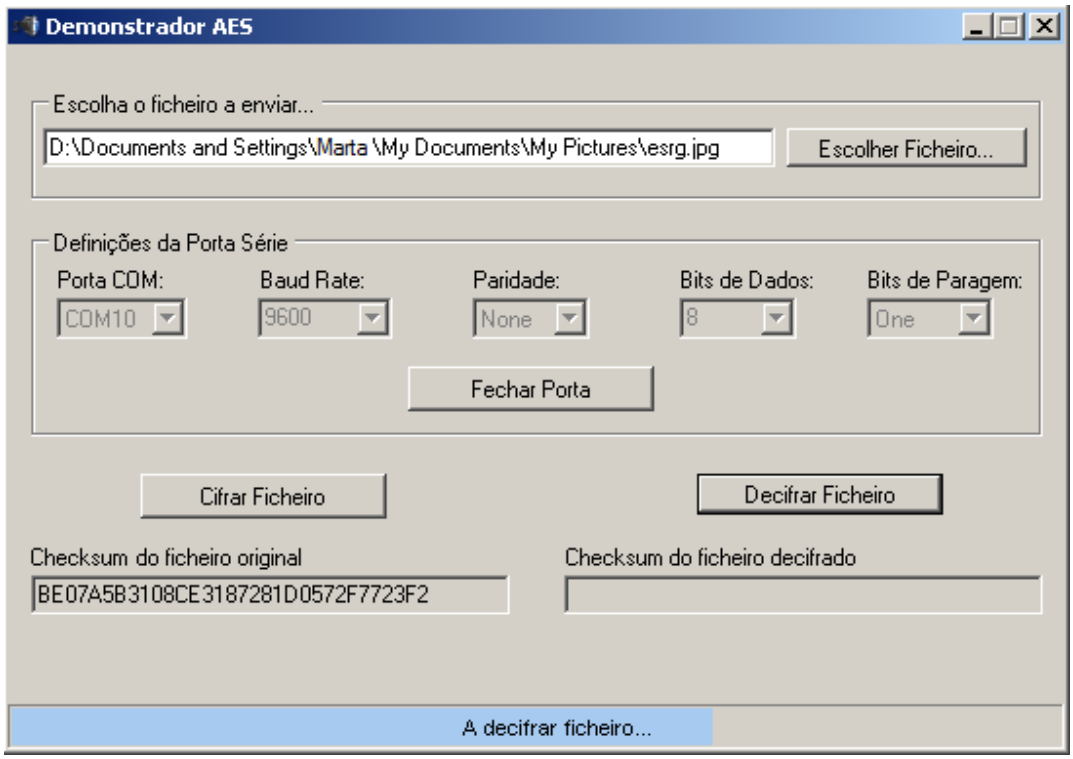


Figura 4.23 – Demonstrador AES a decifrar.

Após a decifragem ocorrer aparece no ecrã uma mensagem de sucesso caso a cifragem e decifragem forem bem-sucedidas. O sucesso é verificado através da comparação do *checksum* do ficheiro original e do ficheiro decifrado, como é possível verificar na figura 4.24.

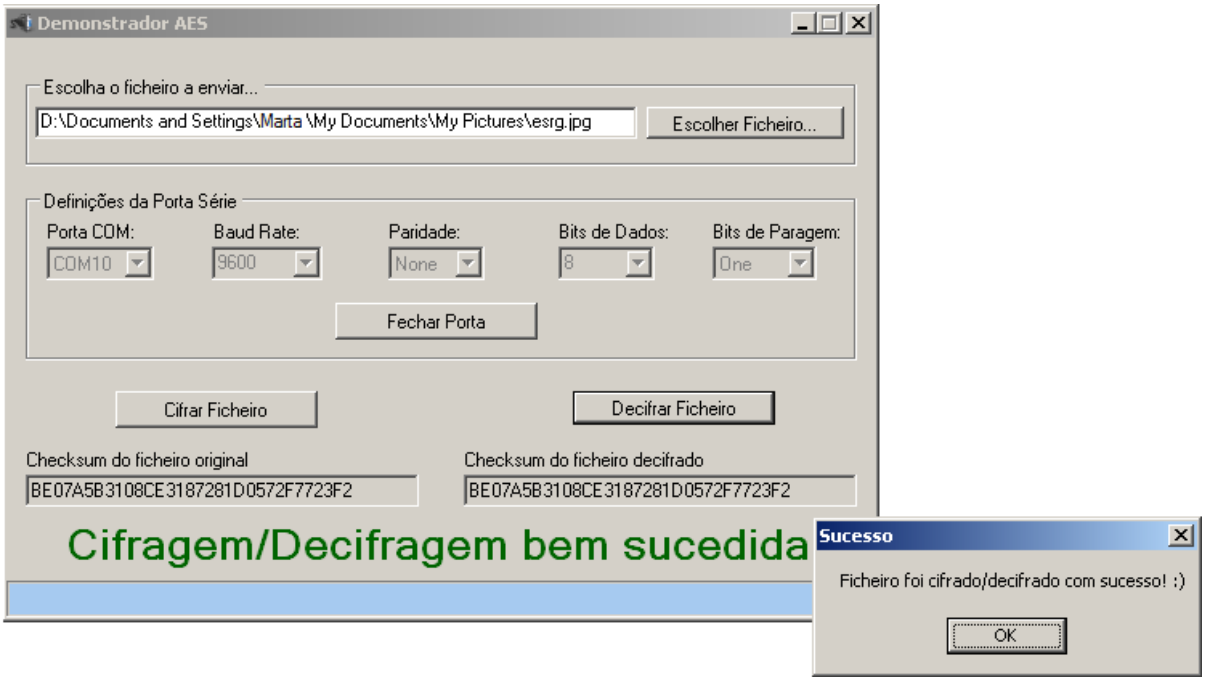


Figura 4. 24 – Cifragem e decifragem concluída com sucesso.

A imagem obtida após a decifragem é a seguinte (figura 4.25):

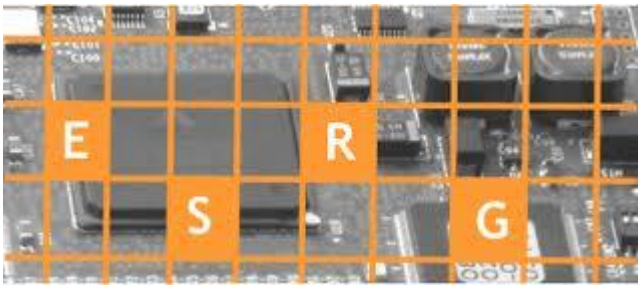


Figura 4. 25 – Ficheiro decifrado.

Como é possível verificar o ficheiro decifrado e o ficheiro original são iguais.

O demonstrador encontra-se no grupo ESRG (*Embedded System Research Group*) no departamento de Engenharia Industrial da Escola de Engenharia da Universidade do Minho, e está disponível para quem desejar testar.

5. Conclusões

A implementação de um SystemVerilog/Verilog pré-processador para lidar com registos TMR, de forma a aumentar o nível de abstração da linguagem SystemVerilog, foi bem-sucedido pois usando as novas metodologias tornam o código HDL muito mais legível e compacto, além de isso é muito simples de usar sempre que se seja usar a técnicas de tolerância a falhas TMR em um sistema basta criar um registo "reg_tmr" e o pré-processador faz todo o trabalho de triplicar os registos e instancia o *voter* para que os registos sejam comparados, devolvendo um ficheiro com toda a lógica implementada e pronto para ser simulado e sintetizado. O caso de estudo implementado do algoritmo de encriptação AES com tolerância a falhas e reimplementado usando as novas metodologias produziram resultados melhores que o esperado o que demonstraram ser boas soluções, no caso do AES com TMR apresentou grandes custos a nível de área ocupada visto que os registos são triplicados o que faz aumentar o *throughput* e consequentemente a frequência máxima usável, por sua vez no caso do AES com Hamming tem menor custo a nível de área mas traz custos a nível de performance, uma vez que as operações demoram N vezes mais tempo a serem realizadas, o *throughput* é inferior que o AES com TMR para o mesmo ciclos de relógio (40 ciclos de relógio). Quando comparados os dois casos de estudos, AES com Hamming e AES com TMR é visível que o AES com TMR produz melhores resultados na correção de erros do tipo SEUs. O caso de estudo implementado com extensão à nova metodologia produziu o mesmo resultado que o AES com TMR mas com a vantagem de requerer um menor esforço de desenvolvimento/engenharia.

Apesar do sucesso do desenvolvimento desta dissertação, e das vantagens que ela oferece, como trabalho futuro pretende-se melhorar o SystemVerilog/Verilog pré processador para lidar com registos Hamming, pretende-se ainda relacionar a fiabilidade com as técnicas de tolerância a falhas, aumentar a capacidade de gerir diferentes aspectos de design através de novas extensões à linguagem.

Pretende-se ainda produzido um artigo científico demonstrando as implementações efectuadas e o seu contributo científico para a área em questão.

Bibliografia

[1] Yijun Liu, Pinghua Chen, Guobo Xie, Zhusong Liu, Zhenkun Li, The Design of a Low-Power Asynchronous DES Coprocessor for Sensor Network Encryption.

[2] Rodrigo Fontes Souto Vasco Roriz, Elias Bechepeche Feliciano de Lima, Implementação do 3DES em Sistemas Embarcados para Terminais de Ponto de Venda.

[3] How-Shen Chang, International Data Encryption Algorithm.

[4] Cifra Sequenciais, José Carlos Bacelar Almeida.

[5] Roohi Banu, Tanya Vladimirova, Fault-Tolerant Encryption for Space Applications.

[6] C.Jeba Nega Cheltha, Prof. R.Velayutham, A Novel Error-Tolerance Method in AES for Satellite Images.

[7] Evangelos Papoutsis, Gareth Howells, Andrew Hopkins, Klaus McDonald-Maier, Key Generation for Secure Inter-satellite Communication.

[8] Anderson, T.; LEE, P. A. Fault tolerance -principles and practice, Englewood Cliffs, Prentice-Hall, 1981.

[9]pgarcia, tgomes, fsalgado, jcabral, paulo.cardoso, atavares, A Fault Tolerant Design Methodology for a FPGA-based Softcore Processor.

[10]M. Fras, H. Kroha, J. v. Loeben, O. Reimann, R. Richter, and B. Weber, Use of Triple Modular Redundancy (TMR) technology in FPGAs for the reduction of faults due to radiation in the readout of the ATLAS Monitored Drift Tube (MDT) chambers.

[11] Gustavo Neuberger, Fernanda de Lima, Luigi Carro, Ricardo Reis, Projeto de uma Memória SRAM Tolerante a Múltiplas Falhas.

[12] Constantin ANTON, Gabriel IANA, Gheorghe SERBAN, Ion TUTANESCU, Petre ANGHELESCU, Hardware Implementation of a Single Bit Error Code Correction.

[13] Shubhendu S. Mukherjee,Christopher Weaver,Joel Emer,Steven K. Reinhardt, Todd Austin, A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor.

[14] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee and Ram Rangan, Computing Architectural Vulnerability Factors for Address-Based Structures.

[15] Mostafa E. Salehi, Ali Azarpeyvand, Farshad Firouzi, Amir Yazdanbakhsh, Reliability Analysis of Embedded Applications in Non-Uniform Fault Tolerant Processors.

[16] Miller-Karlow, D.L.; Golin, E.J.; vVHDL: a visual hardware description language, Visual Languages, 1992. Proceedings., 1992 IEEE Workshop.

[17] American National Standards Institute, IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language, IEEE Std 1364-1995.

[18] Palumbo, G, "Design Through Verilog HDL [Book Review]," Circuits and Devices Magazine, IEEE.

[19] American National Standards InstituteIEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language - Redline.

[20] http://www.systemverilog.org/

[21] William Stalling, Cryptography and Network Security Principles and Practices, Fourth Edition, Prentice Hall, November 16, 2005.

[22] D. Gajski et al. *SpecC: Specification Language andMethodology*. Kluwer, Jan 2000.

Anexo A

Código C – Pré-processador

main.c

```
/*
	Verilog preprocessor to handle Triple Modular Redundancy Registers
*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"parse.h"
#include"symtable.h"

FILE	*input_file;
FILE	*output_file;

int main(int argc, char *argv[])
{
	char	*output_file_name;
	int counter;

	//Checks for correct program arguments
	if(argc!=2)
	{
		printf("Incorrect program usage: <program name><input file>.\n");
		return 0;
	}

	//Attempts to open source file and handles errors
	input_file = fopen(argv[1],"r");
	if(input_file==NULL)
	{
		printf("Could not open input file \"%s\".\n",argv[1]);
		return 0;
	}

	//Attempts to create output file and handles errors
	output_file_name=(char *)malloc(strlen(argv[1]+5));
	strcpy(output_file_name,argv[1]);

	for(counter=0;output_file_name[counter]!='.';counter++)
	{}
	output_file_name[counter]='\0';
	strcat(output_file_name,"_tmr.v");
```

```
	output_file = fopen(output_file_name,"w");
	if(output_file==NULL)
	{
		printf("Could not create output file \"%s\".\n",output_file_name);
		return 0;
	}

	//Parses source file and writes equivalent processed output file
	parse();

	printtable();

	//Closes file handlers
	fclose(input_file);
	fclose(output_file);

	return 0;
}
```

parse.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"parse.h"
#include"symtable.h"

extern FILE	*input_file;
extern FILE	*output_file;

char *get_token(char *token)
{
	int whitespace;


	//Always read one whitespace character, and write to output file
	whitespace=fgetc(input_file);


	if(whitespace==EOF)
	{
		token[0]=EOF;
		return token;
	}
	fprintf(output_file,"%c",(char)whitespace);
	//Read one token
	fscanf(input_file,"%s",token);
```

```
	return token;
}


int get_size(char *token)
{
	int width;
	token++;

	width=token[0]-48;

	token++;
	while(token[0]!=':')
	{
		width*=10;
		width+=token[0]-48;
		token++;
	}
	return width+1;
}

void parse()
{
	char token[200];
	char range[200];
	char var[200];
	char pre_insert[200];
	char *tmp;
	int width;
	FILE *attribution;
	fpos_t pos;

	//To make sure token[0] is not EOF
	token[0]='a';

	//Runs through input file one token at a time untill EOF
	while(!feof(input_file))
	{
		tmp=get_token(token);
		strcpy(token,tmp);
		//If token is "reg_tmr"
		if(strcmp(token,"reg_tmr")==0)
		{
			fprintf(output_file,"reg ");
			//Get new token
			tmp=get_token(token);
			strcpy(token,tmp);

			//If token is size declaration (e.g., [7:0]) writes to output file and gets width
```

```
                        if(token[0]=='[')
                        {
                                fprintf(output_file,token);
                                if(strcmp(token,"endmodule")==0)
                                        return;
                                //Get size from token
                                width=get_size(token);
                                //Get new token
                                tmp=get_token(token);
                                strcpy(token,tmp);
                        }
                        else
                        {
                                width=1;
                        }
                        //Token is variable name
                                strcpy(pre_insert,token);
                                if(pre_insert[strlen(pre_insert)-1]==';')
                                        pre_insert[strlen(pre_insert)-1]='\0';
                                //Adds to symbol table
                                symbol_insert(pre_insert);
                                //Writes expanded declaration to output file
                                if(token[strlen(token)-1]==';')
                                        token[strlen(token)-1]='\0';
                                fprintf(output_file,"%s1, %s2, %s3;\n",token,token,token);
                                fprintf(output_file,"wire [%d:0] %s;\n",width-1,token);
                                //Writes voter instantiation to output file
                                fprintf(output_file,"voter #(%d)
voter_%s(%s,%s1,%s2,%s3);\n",width,token,token,token,token,token);
                                if(strcmp(token,"endmodule")==0)
                                        return;
                }
                //If not "reg_tmr"
                else
                {
                        //If use of symbol table variable
                        if(symbol_match(token))
                        {
                                strcpy(var,token);
                                //Get new token
                                fscanf(input_file,"%s",tmp);
                                strcpy(token,tmp);
                                //If token is range specification
                                if(token[0]=='[')
                                {
                                        //copy range string to range variable
                                        strcpy(range,token);
                                        //Check if attribution
                                        //Get new token
                                        tmp=get_token(token);
```

```
                strcpy(token,tmp);
                if((strcmp(token,"=")==0)||(strcmp(token,"<=")==0))
                {
                        char c='0';
                        //If yes, write equivalent write (with specified range)
to output file
                        fgetpos (input_file,&pos);
                        fprintf(output_file,"%s1%s %s",var,range,token);
                        while(c!=';')
                        {
                                c=fgetc(input_file);
                                fprintf(output_file,"%c",c);
                        }
                        c='0';
                        fsetpos (input_file,&pos);
                        fprintf(output_file,"\n%s2%s %s",var,range,token);
                        while(c!=';')
                        {
                                c=fgetc(input_file);
                                fprintf(output_file,"%c",c);
                        }
                        fsetpos (input_file,&pos);
                        fprintf(output_file,"\n%s3%s %s",var,range,token);
                        if(strcmp(token,"endmodule")==0)
                                return;
                }
                //If not attribution
                else
                {
                        //Just write back variable name and range, plus token
                        fprintf(output_file,"%s%s %s",var,range,token);
                        if(strcmp(token,"endmodule")==0)
                                return;
                }
        }
        //If not range, check if attribution
        else
        {

                //If yes, write equivalent attribution to output file
                if((strcmp(token,"=")==0)||(strcmp(token,"<=")==0))
                {
                        char c='0';
                        //If yes, write equivalent write  to output file
                        fgetpos (input_file,&pos);
                        fprintf(output_file,"%s1 %s",var,token);

                        while(c!=';')
                        {
                                c=fgetc(input_file);
```

```
                                        fprintf(output_file,"%c",c);
                                    }

                                    c='0';
                                    fsetpos (input_file,&pos);
                                    fprintf(output_file,"\n%s2 %s",var,token);
                                    while(c!=';')
                                    {
                                        c=fgetc(input_file);
                                        //printf("%c\n",c);
                                        fprintf(output_file,"%c",c);
                                    }
                                    fsetpos (input_file,&pos);
                                    fprintf(output_file,"\n%s3 %s",var,token);
                                    if(strcmp(token,"endmodule")==0)
                                        return;
                                }
                                //If not attribution
                                else
                                {
                                    //Just write back variable name and token
                                    fprintf(output_file,"%s %s",var,token);
                                    if(strcmp(token,"endmodule")==0)
                                        return;
                                }
                            }
                        }
                        //If not, write token to output file
                        else
                        {
                            fprintf(output_file,token);
                            if(strcmp(token,"endmodule")==0)
                                return;
                        }
                    }
                }
            }
        }
```

parse.h

```
#ifndef PARSE_H
#define PARSE_H

void    parse();

#endif
```

symtable.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"symtable.h"

struct symbol
{
	char *sym_name;
	struct symbol *next;
};
typedef struct symbol symbol;
symbol *symtable=NULL;
void	symbol_insert(char *token)
{
	symbol *tmp;

	//If first symbol, initialize symbol table
	if(symtable==NULL)
	{
		symtable=(symbol *)malloc(sizeof(symbol));
		symtable->next=NULL;
		symtable->sym_name=(char *)malloc(strlen(token)+1);
		strcpy(symtable->sym_name,token);
		return;
	}
	//Not first symbol, run to end of symbol table
	//and create new node
	for(tmp=symtable;tmp->next!=NULL;tmp=tmp->next)
	{}
	tmp->next=(symbol *)malloc(sizeof(symbol));
	tmp=tmp->next;

	tmp->next=NULL;
	tmp->sym_name=(char *)malloc(strlen(token)+1);
	strcpy(tmp->sym_name,token);
	return;
}

int		symbol_match(char *token)
{
	symbol *tmp;

	//If no symbol table, no match
	if(symtable==NULL)
		return 0;

	for(tmp=symtable;tmp!=NULL;tmp=tmp->next)
	{
		if(strcmp(tmp->sym_name,token)==0)
		{
```

```
                        return 1;
                }
        }

        return 0;
}

void printtable()
{
        symbol *tmp;
        printf("Expanded TMR registers:\n");
        for(tmp=symtable;tmp!=NULL;tmp=tmp->next)
        {
                printf("%s\n",tmp->sym_name);
        }
}
```

symtable.h

```
#ifndef SYMTABLE_H
#define SYMTABLE_H

void    symbol_insert(char *token);
int             symbol_match(char *token);
void printtable();

#endif
```